Titre: Title:	Méthode de conception dirigée par les modèles pour les systèmes avioniques modulaires intégrés basée sur une approche de cosimulation
Auteur: Author:	Lin Bao
Date:	2014
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Bao, L. (2014). Méthode de conception dirigée par les modèles pour les systèmes avioniques modulaires intégrés basée sur une approche de cosimulation [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1617/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/1617/
Directeurs de recherche: Advisors:	Guy Bois, & Jean-François Boland
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

MÉTHODE DE CONCEPTION DIRIGÉE PAR LES MODÈLES POUR LES SYSTÈMES AVIONIQUES MODULAIRES INTÉGRÉS BASÉE SUR UNE APPROCHE DE COSIMULATION

LIN BAO

DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAITRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE INFORMATIQUE)

DÉCEMBRE 2014

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

MÉTHODE DE CONCEPTION DIRIGÉE PAR LES MODÈLES POUR LES SYSTÈMES AVIONIQUES MODULAIRES INTÉGRÉS BASÉE SUR UNE APPROCHE DE COSIMULATION

présenté par : BAO Lin

en vue de l'obtention du diplôme de : Maîtrise ès Sciences Appliquées

a été dûment accepté par le jury d'examen constitué de :

Mme NICOLESCU Gabriela, Doctorat, présidente

M. BOIS Guy, Ph.D., membre et directeur de recherche

M. BOLAND Jean-François, Ph.D., membre et codirecteur de recherche

M. LANGLOIS J.M.Pierre, Ph.D., membre

REMERCIEMENTS

Dans un premier temps, j'aimerais remercier mon directeur de recherche professeur Guy Bois, ainsi que mon codirecteur professeur Jean-François Boland, pour leurs soutiens et leurs conseils tout au long de mes travaux de recherche.

Ensuite, je tiens à remercier les partenaires industriels du projet AVIO509 (projet AREXIMAS) CMC Électroniques et CAE Électroniques, le Regroupement Stratégique en Microsystème du Québec (ReSMiQ), le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) et aussi l'École Polytechnique de Montréal pour leurs aides à la fois financière, matérielle et humaine pendant mes travaux de stage et aussi de recherche.

Et puis, j'aimerais aussi remercier tous mes collègues de l'équipe du projet AVIO509, ainsi que les collègues du laboratoire de recherche, pour leurs supports et leurs aides. Je remercie en particulier Julien Savard de son accueil pendant mon stage à l'École Polytechnique de Montréal, ainsi que Jeff FALCON pour son temps et sa patience pour m'aider à la correction grammaticale de ce mémoire.

Enfin, je remercie les autres membres de ma famille (mes parents et mes sœurs) de leurs encouragements et compréhensions.

RÉSUMÉ

Dans l'industrie aérospatiale, le développement des systèmes devient de plus en plus complexe. Dans ce contexte, l'architecture avionique modulaire intégrée (IMA) a été conçue pour remplacer son prédécesseur - l'architecture fédérée, afin de réduire le poids, la puissance dissipée et la dimension des appareils.

Les travaux de recherche présentés dans ce mémoire, dans le cadre du projet en avionique AVIO509, proposent un ensemble de solutions efficaces en termes de temps et de coût pour le développement et la validation fonctionnelle des systèmes IMA.

Les méthodologies présentées se concentrent principalement sur deux flots de conception basés sur : 1) le concept de l'ingénierie dirigée par les modèles et 2) une plateforme de cosimulation.

Dans le premier flot de conception, le langage de modélisation AADL est utilisé pour décrire une architecture IMA, alors que l'environnement OCARINA pour la génération de code certifié (e.g., POK), va être étendu pour permettre la génération de modèles haut niveau simulables par l'environnement commercial SIMA (un simulateur d'applications IMA conforme aux normes ARINC 653). Dans le deuxième flot de conception, Simulink est choisi pour simuler le monde externe du module IMA grâce à la disponibilité d'une libraire de l'avionique qui offre des capteurs et actuateurs, et aussi grâce à son efficacité pour créer les modèles. La plateforme de cosimulation est donc composée de deux simulateurs: Simulink pour la simulation de périphériques et de SIMA pour la simulation des applications IMA. Ceci représente une alternative idéale aux environnements de développement commerciaux, de nos jours très dispendieux. Pour permettre la portabilité, une partition SIMA est réservée pour jouer le rôle d' « adaptateur ». Ici, l'adaptateur est responsable de synchroniser la communication entre les deux simulateurs via le protocole TCP/IP. Seul l'« adaptateur » devra donc être modifié lorsque l'application est portée vers la plateforme de l'implémentation (e.g. PikeOS), puisque la communication interne et la configuration des systèmes restent les mêmes.

À titre d'étude de cas, une application avionique a été développée dans le but de démontrer la validité des flots de conception.

La recherche proposée dans ce mémoire est une continuité de projets de l'équipe de recherche AVIO 509, et parallèlement pourra être étendue dans le cadre des travaux futurs.

ABSTRACT

In the aerospace industry, with the development of avionic systems becomes more and more complex, the integrated modular avionics (IMA) architecture was proposed to replace its predecessor – the federated architecture, in order to reduce the weight, power consumption and the dimension of the avionics equipment.

The research work presented in this thesis, which is considered as a part of the research project AVIO509, aims to propose to the aviation industry a set of time-effective and cost-effective solutions for the development and the functional validation of IMA systems.

The proposed methodologies mainly focus on two design flows that are based on: 1) the concept of model-driven engineering design and 2) a cosimulation platform.

In the first design flow, the modeling language AADL is used to describe the IMA architecture. The environment OCARINA, a code generator initially designed for POK, was modified so that it can generate avionic applications from an AADL model for the simulator SIMA (an IMA simulator compliant to the ARINC653 standards). In the second design flow, Simulink is used to simulate the external world of IMA module thanks to the availability of avionic library that can offer lots of avionics sensors and actuators, and as well as its effectiveness in creating the Simulink models. The cosimulation platform is composed of two simulators: Simulink for the simulation of peripherals and SIMA for the simulation of IMA module, the latter is considered as an ideal alternative for the super expensive commercial development environment. In order to have a good portability, a SIMA partition is reserved as the role of « adapter » to synchronize the communication between these two simulators via the TCP/IP protocol. When the avionics applications are ported to the implementation platform (such as PikeOS) after the simulation, there is only the « adapter » to be modified because the internal communication and the system configuration are the same.

An avionics application was developed as a case study, in order to demonstrate the validation of the proposed design flows.

The research presented in this paper is a continuation of project of the AVIO509 research team, and parallelly may be extended in the future work.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

La liste des sigles et abréviations présente, dans l'ordre alphabétique, les sigles et abréviations utilisés dans le mémoire ou la thèse ainsi que leur signification.

AADL	Architecture Analysis and Design language
ADL	Architecture Description Language
APEX	ARINC653 Application Executive
API	Application Programming Interface
AREXI MAX	Architectural Exploration in Integrated Modular Avionics Systems
AUTOSAR	AUTomotive Open System ARchitecture
BSP	Board Support Package
COESA	Committee on Extension to the Standard Atmosphere
COTS	Commercial-Off-The-Shelf
CRIAQ	Consortium de Recherche et d'Innovation en Aérospatiale au Québec
GPS	Global Positioning System
IDE	Integrated Development Environment
IMA	Integrated Modular Avionics
IP	Internet Protocol
FIFO	First In First Out
MDE	Model Driven Engineering
MOS	Module Operating System
MTF	Major Time Frame
OS	Operating System
OSI	Open Systems Interconnection
PME	Petites et Moyennes Entreprises

POS	Partition Operating System
POSIX	Portable Operating System Inteface
RTE	Run-Time Environment
ROM	Read-Only Memory
RTOS	Reel Time Operating System
SIMA	Simulated Integrated Modular Avionics
SWaP	Space, Weigh and Power
SysML	Systems Modeling Language
TCP	Transmission Control Protocol
TSP	Time and Space Partitioning
UML	Unified Modeling Language
XML	Extensible Markup language

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Ce chapitre présente le contexte, la problématique, les objectifs, la méthodologie et la contribution de ce projet de maîtrise.

1.1 Contexte du projet et problématique

Afin d'éviter une catastrophe potentielle dans le domaine de l'avionique, aucune défaillance n'est tolérée pendant la durée de fonctionnement d'un appareil. Pour ces systèmes avioniques, la sécurité est donc essentielle. En anglais, on utilise le terme bien connu de « Safety-critical» que nous traduirons ici par système critique. Un *système critique* est un système dont la panne peut avoir des conséquences dramatiques (e.g., mortelles). Les systèmes avioniques doivent donc obligatoirement respecter des règles très strictes.

1.1.1 Architecture IMA

L'architecture des systèmes modulaires intégrés, nommée architecture IMA (Integrated Modular Avionics) [2], a été conçue pour l'industrie aérospatiale comme la nouvelle génération d'architecture avionique. Elle est considérée comme le successeur de l'architecture fédérée pour résoudre des problèmes de dimension, poids et puissance, de l'anglais SWaP (Size, Weigh and Power), dans un contexte où les systèmes avioniques deviennent de plus en plus complexes.

Dans l'architecture fédérée, chaque application possède sa propre plateforme matérielle, la règle principale [1] est: "un calculateur par application", alors que l'architecture IMA propose un seul système intégré et partitionné. Le mot « partitionné » désigne ici créer des partitions (conteneurs d'applications) spatiales et temporelles à l'aide d'un système d'exploitation (en anglais on dit *Time and Space Partitioning Operating System*). Cette classe de systèmes d'exploitation servant d'environnement d'exécution à l'architecture IMA respecte la norme ARINC653 [4 - 6]. ARINC 653 est donc une norme de partitionnement où chaque partition a son propre espace mémoire et temporel, la ségrégation étant assurée par l'OS. La communication entre les modules IMA se fait souvent par un bus AFDX, qui lui est défini par la norme ARINC664 [7]. En résumé, il y a deux principes [1] importants à respecter dans ce type de conception: un principe *d'intégration* de plusieurs applications partageant les ressources sur une même plateforme matérielle IMA et un principe de *certification*. Malheureusement, l'union de ces deux principes rend très complexe et

très dispendieux le processus de développement et de certification des systèmes IMA [1]. Finalement, notons que le concept d'architecture IMA est proposé depuis des années et il a servi à la réalisation des avions de ligne Airbus380 et Boing787 [1]. Il existe plusieurs OS commerciaux qui supportent les normes ARINC653 et ARINC664, par exemple PikeOS de Sysgo, VxWorks653 de Wind River. Ces OS et leurs environnements de développement (simulation, déverminage, traçage, analyse de performance, etc.) sont complexes et leur prix est généralement élevé. Plusieurs équipes de développement logiciel hésitent ou ne peuvent simplement pas (cas de la petite et moyenne entreprise) investir une telle somme d'argent pour acquérir ce type d'outils.

1.1.2 L'ingénierie dirigée par les modèles

La modélisation et la simulation sont reconnues comme deux étapes très importantes pour la conception des systèmes embarqués. Ils permettent de valider la spécification et de vérifier que la spécification correspond bien aux différents modèles générés lors de son raffinement. Il est démontré qu'un flot de conception dirigé par des modèles, dont l'utilisation intermédiaire d'une plateforme de simulation fonctionnelle avant l'étape de l'implémentation sur matériel, diminue significativement le processus de développement (e.g. temps de développement et coût).

Le développement dirigé par les modèles consiste tout simplement à construire un modèle pour un système, pour que ce modèle puisse être transformé en quelque chose de réel [12] tel que du code C/C++. Un modèle est un ensemble d'éléments visant à décrire un système complexe. De nos jours, cette méthodologie a gagné en popularité. Il existe des outils de modélisation très variés pour aider les développeurs ou concepteurs des applications, systèmes, architectures et matériaux (UML, AADL, SysML, etc). Il est moins coûteux (en temps) d'écrire une ligne en C++ que d'écrire plusieurs lignes en assembleur, parallèlement, il est moins coûteux de construire un modèle en UML que de programmer plusieurs lignes en C++. Le développement dirigé par les modèles est souvent accompagné de génération de code grâce aux techniques d'automatisation permettant le passage d'un niveau d'abstraction à un autre niveau d'abstraction, comme par exemple le passage d'un modèle UML à du code C++. L'ingénierie dirigée par les modèles est considérée aujourd'hui comme une solution très efficace pour développer les systèmes avioniques critiques. Il est reporté que 70% des erreurs sont faites pendant le processus de

spécification et avant l'effort de l'implémentation [17]. Cette méthodologie permet de détecter et analyser les erreurs dès le début du cycle de vie du logiciel. Donc, en résumé, l'ingénierie dirigée par les modèles diminue le coût de développement et augmente la productivité du développement. Dans ce projet, nous nous intéressons à un environnement commercial nommé SIMA, un simulateur fonctionnel d'applications IMA conforme aux normes ARINC 653. Nous nous intéresserons à la problématique du passage du langage de spécification AADL à un modèle fonctionnel pour une simulation exécutable sur SIMA, pour ensuite poursuivre le raffinement du modèle vers l'implémentation.

1.1.3 Les modèles de cosimulation

Les systèmes avioniques, tels que le système de contrôle de vol, le système de carburant et le système de navigation, sont eux-mêmes constitués de périphériques tels que des capteurs (thermiques, pressions, etc.), des actuateurs (valves, moteurs électriques, etc.). Ces périphériques sont essentiels pour générer les bons stimuli d'un banc de test lors d'une simulation. Toutefois, ces matériaux d'avionique coûtent très cher et il est très difficile de les introduire directement dans une simulation. Une solution utilisée par l'industrie est de créer des modèles en logiciel simulant les fonctionnalités des composants en matériel (vitesse, altitude, vent, etc.). Des outils de modélisation comme Simulink de Matlab [9], Labview de National Instruments [38] permettent alors de développer rapidement des modèles standards de capteurs et actuateurs. Lors de la phase d'implémentation, ces modèles de périphériques peuvent être substitués par leur équivalent matériel.

La Figure 1-1 montre une structure commune d'une plateforme de cosimulation qui sert à faire une simulation conjointe entre différents composants du système, chaque composant ayant son propre simulateur (modèle de simulation). Cette méthode est souvent utilisée par les concepteurs pour simuler des systèmes complexes. La cosimulation permet d'accéder à des librairies de différents outils de développement, donc la capacité de simulation est évidemment augmentée. Il est possible de décrire et simuler ensemble différents niveaux d'abstraction. Une fois les interfaces de cosimulation bien définies, il est alors plus facile de découper les tâches pour les différentes équipes de conception.

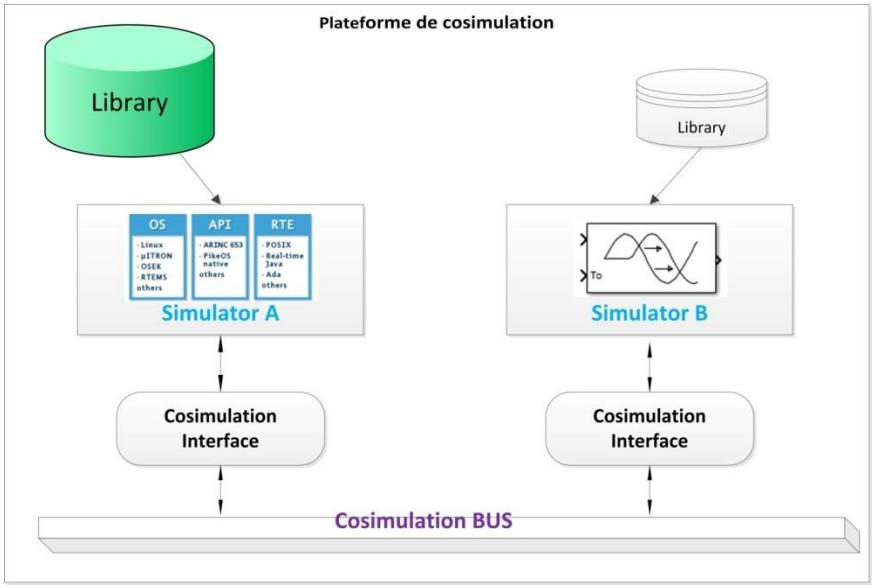

Figure 1-1: Plateforme de cosimulation

SIMA et Simulink représentent un exemple intéressant de deux simulateurs complémentaires pouvant servir à la cosimulation fonctionnelle d'un système IMA : SIMA simule le comportement du système (e.g., traitement), alors que Simulink simule celui des interfaces (périphériques) du système, donc la communication avec le monde externe au système.

1.1.4 Projet Avio 509

Le projet CRIAQ Avio 509, aussi nommé AREXIMAX [3], est un projet de recherche universitaire dans le cadre du consortium de recherche CRIAQ, d'une subvention de recherche et développement coopérative du Canada (CRSNG) et d'une subvention MITACS, en collaboration avec Polytechnique Montréal, École de Technologie Supérieure, CMC électronique Inc. [10] et CAE. Inc. [11].

Le but de ce projet consiste à faire une exploration architecturale sur les systèmes avioniques IMA et de produire une main-d'œuvre spécialisée pour le domaine de l'avionique. Ce projet a été lancé officiellement en juin 2011. Le lecteur pourra trouver plus de détails sur le projet Avio 509 en visitant le site web du projet [3].

Les objectifs du projet de recherche présenté dans ce mémoire font partie des objectifs de recherche du projet Avio 509.

1.2 Objectifs du projet de recherche

Les travaux de recherche réalisés dans ce mémoire visent principalement à définir un flot de conception pour système IMA allant d'une spécification AADL à l'implémentation, en passant par un modèle fonctionnel.

À son tour, cet objectif général peut se diviser en cinq objectifs spécifiques:

i. Permettre la traduction automatique d'une spécification système décrite en AADL à une architecture pour la simulation fonctionnelle d'un système avionique modulaire respectant les normes ARINC 653.

ii. Proposer un modèle de cosimulation afin d'inclure au modèle fonctionnel SIMA, un ensemble de périphériques (capteurs, acteurs, etc.). Cela permet donc la simulation de système IMA plus complet (traitement et communication externe).

iii. Traduire le modèle de cosimulation en implémentation.

iv. Démontrer que cet ensemble de solutions fait baisser les coûts et le temps de la conception.

v. Réaliser une étude de cas pour valider le flot de conception proposée.

1.3 Méthodologie

À partir des cinq objectifs spécifiques, voici les grandes lignes de la méthodologie proposée dans ce travail :

i. Le langage AADL a été choisi pour modéliser le système. Puis un outil de traduction du domaine public nommé OCARINA générera une description servant à construire le modèle fonctionnel. SIMA sera utilisé pour faire la validation et la vérification fonctionnelle. La génération automatisée implique de fournir : 1) une entrée, 2) une configuration et 3) un API ARINC 653 à SIMA. Notez que le choix d'AADL, OCARINA et SIMA a été imposé par le succès du projet précédent, réalisé par un autre membre de l'équipe de recherche [8] (voir détails à la section 1.4-1).

ii. L'outil "Simulink" a été sélectionné grâce à la grande disponibilité de ses librairies et aussi grâce à sa facilité pour effectuer le développement de modèles de périphériques avioniques. Du code C/C++ sera généré à partir de ces modèles, puis sera intégré dans une partition du système.

iii. Au niveau de l'implémentation, le logiciel PikeOS de la société Sysgo est utilisé comme OS. Puisqu'aucun portage d'OS n'est actuellement disponible pour une carte matérielle utilisée en avionique (idéalement un processeur Freescale), nous avons ciblé un processeur Intel d'usage général (x8086). Plus précisément, l'architecture matérielle sera simulée sous QEMU qui est un logiciel libre de machine virtuelle. Cet émulateur peut simuler un processeur ou une architecture [41]. Pour assurer une bonne portabilité vers PikeOS, le module IMA et ses périphériques seront séparés et communiqueront via un protocole de communication tel que TCP/IP. Le système IMA sera constitué d'une partition pour la communication et d'une (ou plusieurs) partition(s) applicative(s). La partition communication qui jouera le rôle d'adaptateur, fera l'interface entre les périphériques et le reste du système. L'avantage de cette architecture est qu'une fois la simulation fonctionnelle sous SIMA complétée, les applications validées pourront être réutilisées directement par PikeOS, il ne restera que l'adaptateur à modifier. Plus précisément, dans une dernière étape qui dépasse le cadre de ce projet, les périphériques simulés pourraient être remplacés directement par les vrais modèles correspondants et la communication entre le module IMA et ses périphériques pourrait être réalisée par un bus ARINC429 ou AFDX.

iv. La motivation derrière la méthodologie proposée dans ce projet vient du fait qu'une simulation intermédiaire de type fonctionnel peut faire baisser considérablement le coût d'une conception, car elle permet de déverminer le système tôt dans le processus de conception. En effet, une erreur détectée suite à une mauvaise interprétation de la spécification sera toujours plus coûteuse lorsqu'elle est détectée à l'implémentation plutôt qu'aux premiers niveaux de conception. De plus, le choix des outils pour ce projet rend ce flot de conception abordable. À titre d'exemple, le logiciel OCARINA appartient au domaine public, alors que le coût d'une licence du logiciel SIMA est de l'ordre de quelques milliers de dollars. Nous aurions souhaité explorer les possibilités

d'utiliser un OS du domaine public (tel que exemple POK [39]), mais cela dépassait le cadre du présent projet. Il nous est donc apparu plus simple d'utiliser un OS commercial établi tel que PikeOS pour compléter notre preuve de concept.

v. Finalement, une étude de cas sera réalisée pour démontrer la rapidité et la facilité pour développer des systèmes avioniques en utilisant des flots de conception, la portabilité des applications avioniques d'un simulateur IMA vers un vrai OS partitionné PikeOS. Cet exemple va montrer que l'ensemble de solutions proposé est efficace en termes de temps et aussi en termes de coût.

1.4 Contributions

Cette section décrit trois contributions importantes réalisées dans le cadre de cette recherche:

1. La génération automatique d'un modèle IMA fonctionnel et simulable sur SIMA à partir de la combinaison des deux travaux de recherche. En effet, mentionnons que dans le cadre des travaux d'un autre membre du projet [8] (Julien Savard), une étude théorique sur la faisabilité d'introduire un flot de conception basé sur les modèles a été présentée. La partie mise en œuvre qui constitue des ajouts au générateur OCARINA n'a toutefois pas été complétée par M. Savard, car elle fait partie du présent travail de recherche. Les résultats des travaux théoriques et de leur mise en œuvre ont conduit à une présentation et publication à une conférence de SAE[1] [29].
2. La proposition d'une plateforme de cosimulation SIMA/Simulink ciblant principalement des actuateurs et des capteurs pour le domaine de l'avionique. Ces derniers ont été développés pour les besoins du présent projet, c'est-à-dire fournir des périphériques de base pour un environnement de vol, à partir de primitives existantes dans la librairie Simulink. Notez qu'au meilleur de notre connaissance aucun travail proposant cette cosimulation SIMA/Simulink n'a été proposé dans la littérature à ce jour. Notre approche propose donc l'interopérabilité entre les deux outils de simulation.

[1] SAE: Society of Automotive Engineers

3. Le raffinement du modèle de cosimulation vers une implémentation pour système avionique modulaire (IMA) basé sur l'OS PikeOS.

En résumé, à partir d'une description AADL et à l'aide du logiciel OCARINA, nos contributions permettent de générer automatiquement un modèle IMA complet (traitement et communication externe) fonctionnel et conjointement simulable avec SIMA et Simulink, puis de finalement offrir un raffinement automatisé pour une implémentation IMA. À la fin, il ne reste que l'adaptateur à modifier afin de remplacer les périphériques en logiciel directement par les périphériques physiques correspondants. Tel que mentionné plus haut, une communication réalisée par un bus ARINC429 ou AFDX entre le module IMA et ses périphériques pourra éventuellement être ajoutée.

Dans la suite de ce travail, le chapitre 2 présente la revue de littérature, le chapitre 3 présente en détail la méthodologie qui permet de rencontrer nos objectifs spécifiques, le chapitre 4 présente une étude de cas et le chapitre 5 fournit les résultats obtenus. Enfin, la conclusion et les travaux futurs sont présentés.

CHAPITRE 2 REVUE DE LITTÉRATURE

Ce chapitre présente la revue de littérature en lien avec ce projet de recherche. Il se concentre sur les architectures IMA et leurs simulations, le système d'exploitation PikeOS, les langages de modélisation et les bus de communication avionique.

2.1 IMA

Pour des raisons de sûreté, pendant des années, les systèmes avioniques étaient conçus pour exécuter une seule application sur une plateforme dédiée en ciblant une architecture où le matériel et le logiciel étaient couplés. Tel que mentionné précédemment cette architecture porte le nom « d'architecture fédérée ».

Afin de respecter les priorités de l'industrie de l'aviation en matière de dimension, poids et puissance (Swap), la nouvelle génération d'architecture abrégée par IMA (de l'anglais « Integrated Modular Avionics »), nous permet l'utilisation de partitions. Une partition est un conteneur d'une (partie de l') application qui contient du code, des données et de l'information sur la configuration courante. Le partitionnement implique que les applications avioniques soient séparées en termes de temps et d'espace. La séparation temporelle signifie que toutes les applications avioniques sont exécutées dans un ordre prédéfini et durant un délai prédéfini, autrement dit, pour un instant donné il n'y a qu'un seul programme qui s'exécute. Par conséquent, il n'existe pas de compétition temporelle entre les différentes applications avioniques tout au long de la durée d'exécution. D'autre part, la séparation spatiale signifie que chaque partition possède sa propre mémoire protégée. Une application ne peut donc pas directement accéder à des adresses en dehors de cette plage d'adresse et venir ainsi corrompre les données d'une autre application. Bref, ce mécanisme de protection de mémoire garantit qu'il n'y a pas de possibilité de conflit de mémoire.

Deux niveaux d'ordonnancement son utilisés: 1) un ordonnancement *inter-partition* et 2) un ordonnancement *intra-partition*. L'ordonnancement inter partition implique que les partitions s'exécutent périodiquement en respectant un ordonnancement statique qui spécifie l'ordre et la durée d'exécution. Lorsqu'une partition est ordonnancée, un programme à son tour composé d'un ou de plusieurs processus peut alors s'exécuter. Un processus est un code logiciel (tâche) qui s'exécute en concurrence avec les autres processus de la partition. Quand le temps d'exécution lié

à une partition expire, son programme est préempté, ce qui implique que la prochaine fois il poursuivra son exécution à partir du point d'arrêt. Tel que mentionné plus haut, une même partition peut contenir un ou plusieurs processus. Ces processus sont ordonnancés par un mécanisme de priorité. Pour les processus de même priorité, on respecte la règle du *premier arrivé, premier servi*. [13]. De cette façon, on peut exécuter plusieurs applications concurremment sur la même plateforme. Ces applications sont indépendantes les unes par rapport aux autres grâce à la séparation du système. Donc, la défaillance d'une partition n'aura pas d'effet sur une autre partition.

Le partage de ressources et le partitionnement du système sont deux caractéristiques principales de l'architecture IMA [1]. Ces deux principes sont spécifiés respectivement par deux standards: ARINC653 [4-6] et ARINC664 [7]. Le standard ARINC653 explique en détail comment gérer plusieurs applications avioniques s'exécutant sous le même OS. Le standard ARINC664 se concentre sur les mécanismes de communication entre les différents modules.

La figure 2-1 présente le modèle de l'architecture IMA. Au-dessus de la plateforme matérielle et de la couche d'abstraction du matériel (abrégé par BSP, de l'anglais *Board Support Package*), il existe deux couches d'OS: le MOS (Module OS) et le POS (Partition OS). Le MOS s'occupe de l'ordonnancement des partitions, le service de communication inter-partition et la gestion de signaux de la plateforme matérielle [2]. Le POS, qui peut changer d'un partitionnement à l'autre (Linux, VxWorks, uC, etc.), est une partie du système [2], il se charge de l'ordonnancement des processus et des données qui composent une même partition et aussi la gestion de ressources de la partition.

L'ARINC653 définit un API[2] standard et des services pour IMA, soit la couche APEX (ARINC653 Application Executive). Grâce à l'APEX, le MOS assure un partitionnement temporel et spatial permettant ainsi aux applications avioniques d'avoir une bonne portabilité, ce qui favorise aussi la réutilisation du code et des données. Par conséquent, le coût, le temps de développement et l'effort d'intégration pour les systèmes avioniques peuvent être grandement diminués. Encore une fois, par rapport à l'architecture fédérée, l'architecture IMA permet que

[2] API (pour Application Programming Interface) : c'est un ensemble normalisé de classes, de méthodes ou de fonctions qui sert de façade par laquelle un logiciel offre des services.

plusieurs applications/partitions partagent la même plateforme matérielle, ce qui économise la quantité de matériel utilisée [2].

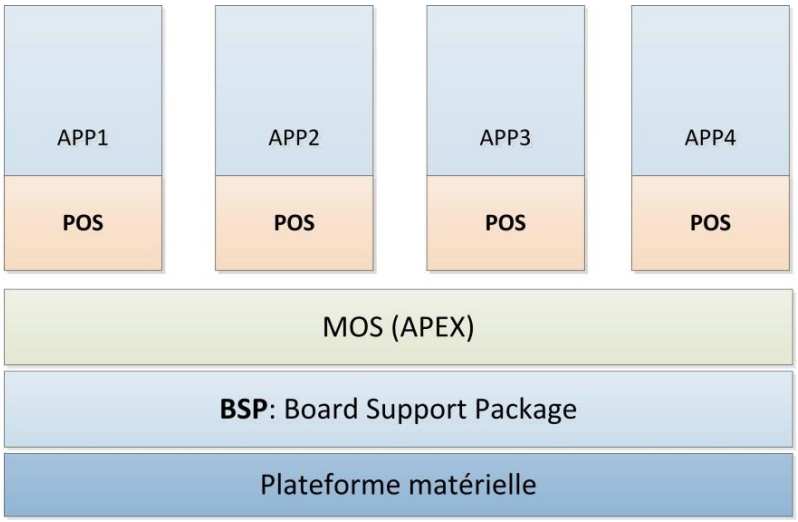

Figure 2-1: l'Architecture IMA

2.1.1 ARINC653 et APEX portable

Le standard ARINC653 définit clairement l'interface et le comportement de l'APEX. L'objectif principal de la couche APEX est de proposer aux applications avioniques des services définis et d'offrir une bonne compatibilité et portabilité aux applications, en fait, l'interface APEX fournit les services suivants [4-6] au POS :

1. Gestion de partition
2. Gestion de processus
3. Gestion du temps
4. Gestion de la communication intra-partition
5. Gestion de la communication inter-partition
6. Gestion des pannes
7. Gestion de la mémoire

2.1.1.1 Gestion de partition

Le MOS se charge du partitionnement du système et la gestion des partitions à l'intérieur de ce module. Cette couche d'OS doit être strictement isolée et protégée pour éviter la défaillance potentielle du logiciel dans la partition.

Les partitions sont ordonnancées de façon statique et cyclique. L'OS d'un module définit une période principale (abrégé par MTF de l'anglais Major Time Frame) d'une durée fixe. Les temps d'exécution alloués aux partitions sont représentés par une série de fenêtres placées dans la période principale. L'ordre de ces fenêtres est aussi prédéfini. Chaque partition est définie par une compensation (« offset ») par rapport au début de la période principale MTF et d'une durée d'exécution prédéfinie [4]. Les durées d'exécution des partitions peuvent être différentes entre elles et être déterminées selon le besoin du code qui les compose. La période principale MTF doit donc être supérieure à la somme des durées d'exécution de toutes les partitions.

2.1.1.2 Gestion de processus

Un processus contient du code exécutable, des données, une pile d'exécution, les pointeurs de piles, le compteur du programme et d'autres attributs [4]. Comme précisé plus tôt, dans une même partition, plusieurs processus travaillent à fournir une fonctionnalité bien précise. Ces processus-ci partagent la même mémoire. Lors de la définition d'un processus, certains attributs fixés doivent être spécifiés : le nom de processus, son point d'entrée, la taille de sa pile, sa priorité de base, sa période, son temps d'exécution et l'échéance (ou en anglais *deadline*). Les processus d'une partition sont créés et initialisés lors de l'initialisation de la partition. C'est la partition qui s'occupe de la gestion du comportement des processus la constituant.

2.1.1.3 Gestion du temps

L'OS dit *partitionné* pour le domaine de l'avionique est un type de système d'exploitation temps réel pour lequel la gestion du temps est une caractéristique importante. Dans le standard ARINC653, la définition du temps dans un contexte IMA est la suivante [4] : "Le temps est unique et indépendant de l'exécution de la partition dans un module. Toutes les valeurs temporelles sont liées à ce moment unique et ne dépendent pas de l'exécution d'une autre partition".

La gestion du temps de la couche APEX fournit une base temporelle unique. Ce sont des tranches de temps pour des fonctionnalités telles que l'ordonnancement de partitions, l'échéance, le retard pour l'ordonnancement de processus, etc. Dans ce contexte, le gestionnaire de temps offre des services [13] pour assurer la cohérence des processus au niveau temporel: mise en attente d'un processus pendant un temps donné, mise en attente d'un processus périodique jusqu'à sa prochaine période d'exécution, etc.

2.1.1.4 Gestion de la communication intra-partition

Pour cette partie, l'APEX fournit des mécanismes pour la communication et la synchronisation entre les différents processus s'exécutant dans la même partition. Plusieurs méthodes sont accessibles : par message, tampon, tableau noir, événement et sémaphore [4]. Quand les ressources de communication demandées par un processus sont temporairement indisponibles, ce processus est bloqué. S'il y a plusieurs processus qui demandent ces ressources, ces processus sont mis dans une liste d'attente selon leur priorité (mode préemptif) ou dans un ordre « premier arrivé, premier servi » (FIFO) [13]. Un processus est retiré de la liste lorsqu'il arrive en tête de liste ou quand il arrive à l'expiration de son temps d'attente (*time-out*).

2.1.1.5 Gestion de la communication inter-partition

Également, l'APEX fournit des mécanismes pour la communication et la synchronisation entre les différents processus s'exécutant dans des partitions différentes. La communication inter partition est réalisée par l'échange de messages. Il y a principalement deux méthodes : 1) par tampon et 2) par échantillonnage. Pour la première méthode, le processus expéditeur met des messages dans une mémoire tampon et le processus destinataire va y chercher ces données. Pour la deuxième méthode, les données sont échangées à travers un canal.

2.1.1.6 Gestion des pannes

Pour mieux tracer et contrôler les erreurs potentielles, ARINC 653 définit des fonctions pour répondre aux erreurs ou défaillances potentielles, dans le but d'éviter des catastrophes. Les erreurs sont classées en trois niveaux : niveau de processus, niveau de partition et niveau de module. ARINC 653 propose un tableau de configuration permettant de spécifier ces erreurs de différents niveaux et des traitements correspondants.

2.1.1.7 Gestion de la mémoire

Une partition est libre de sa propre gestion mémoire. Aucun service n'est spécifié à cette fin.

2.1.2 SIMA

2.1.2.1 Description

SIMA (de l'anglais Simulated Integrated Modular Avionics) est un environnement d'exécution qui supporte ARINC653 pour les systèmes d'exploitation qui ne supportent pas le mécanisme de partitionnement. Il est considéré comme un simulateur IMA qui fonctionne sur tous les OS supportant POSIX tel que Linux/GNU. POSIX (Portable Operating System Inteface) définit un API pour assurer la compatibilité entre Linux/GNU et les autres systèmes d'exploitation [14].

Comme précisée plus tôt, une partition est un conteneur d'applications (processus) ayant son propre bloc de mémoire protégé. Dans POSIX, il y a aussi deux types d'objets : processus et fil d'exécution (thread). Une comparaison entre APEX et POSIX est présentée au tableau 2.1. Le processus sous l'environnement POSIX est similaire à la partition de l'APEX, alors que le fil d'exécution du POSIX est similaire au processus de l'APEX. Sous SIMA, un lien (assignation) existe donc entre l'APEX et le POSIX: une partition d'APEX est représentée par un processus et un processus d'APEX est représenté par un fil d'exécution.

Tableau 2.1: APEX vs POSIX

	Sans protection de mémoire	**Avec protection de mémoire**
APEX (ARINC653)	Processus	Partition
POSIX	Fil d'exécution (thread)	Processus

Sous SIMA, les partitions sont donc ordonnancées par le noyau de Linux. Linux n'étant pas vraiment un OS partitionné, le noyau de Linux ne fait pas le partitionnement temporel et la gestion de panne. Mais, une partition spéciale est réservée pour simuler le comportement d'un vrai MOS et elle s'occupe alors du partitionnement temporel et la gestion de panne. Un ordonnanceur est aussi implémenté dans cette partition.

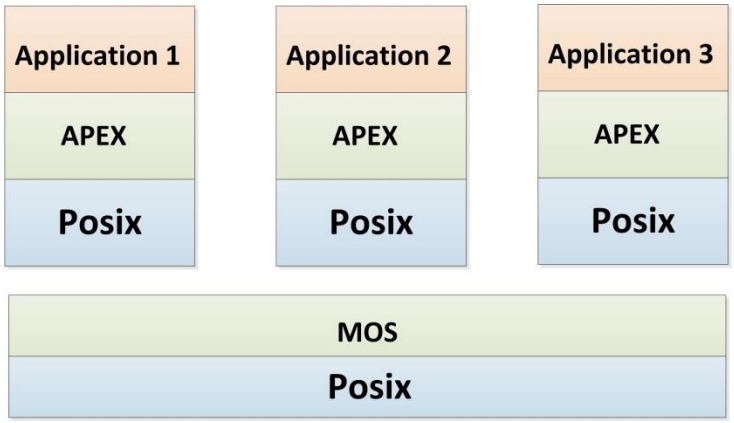

Figure 2-2: l'Architecture SIMA

Au-dessus de la couche POSIX, l'ordonnanceur de processus est implémenté par le POS de SIMA dans le but de gérer les différents processus dans une même partition. L'avantage de cette séparation de l'APEX et l'OS est évident: conduire à une plus grande portabilité. De cette façon, SIMA peut s'exécuter sur tous les systèmes d'exploitation supportant POSIX. Afin de bien gérer les partitions, tel que la réinitialisation ou la suspension, des commandes doivent être échangées entre le MOS et le POS, via les mécanismes de signaux et de partage de mémoire [14].

Comme un simulateur fonctionnant sur un autre OS, SIMA est capable de simuler une grande partie du comportement de l'architecture IMA, tout en étant indépendant de la plateforme matérielle spécifique. Le POS de SIMA respecte les règles essentielles à la sécurité (*safety critical*) [14], mais puisque le MOS de SIMA s'exécute sur un OS non fiable tel que Linux, quand Linux éprouve des problèmes, les applications avioniques ne s'exécutent plus correctement. Toutefois, SIMA permet de simuler et de valider efficacement un modèle fonctionnel haut niveau d'une application IMA respectant la norme ARINC 653.

2.1.2.2 Environnement de développement

D'après nos études, il n'existe pas encore d'environnement disponible pour aider à développer des applications sous le simulateur SIMA. Cependant, le fournisseur offre un grand nombre d'exemples d'application qui peuvent servir de référence. Il est donc possible de créer une application à partir d'un gabarit fourni.

Quand le code d'usager (en C/C++) est prêt, il faut encore configurer le système et le simulateur pour effectuer la simulation d'un système sous SIMA. La configuration du système à simuler est

faite via un fichier XML conforme à la norme ARINC653. En ce qui concerne la configuration du simulateur, un autre fichier XML doit être créé pour spécifier des informations concernant le simulateur. Aussi, les fichiers de code script nécessaires pour lancer les applications et les fichiers de compilation doivent être définis. Ces travaux de configuration peuvent être faits manuellement à partir d'un gabarit, ils peuvent être réalisés aussi par un outil de configuration appelé CONFIGUIMA [44]. Ce dernier est un greffon d'Eclipse permettant de configurer les systèmes ARINC653 via une interface graphique et de générer automatiquement les fichiers XML. Cet outil permet aussi de faire une vérification automatique sur les configurations de système.

2.1.3 PikeOS

2.1.3.1 Description

Le logiciel PikeOS, développé par la compagnie Sysgo [16], est une plateforme virtuelle visant à développer des systèmes embarqués complexes où plusieurs OS et applications s'exécutent simultanément dans un environnement d'exécution avec un haut niveau de sécurité [15]. Cette plateforme supporte le partitionnement temporel et le partitionnement spatial. La figure 2-3 illustre l'architecture de PikeOS. Entre la plateforme matérielle et les partitions, il existe deux couches additionnelles: le micronoyau de PikeOS et l'API PSSW (PikeOS System Software).

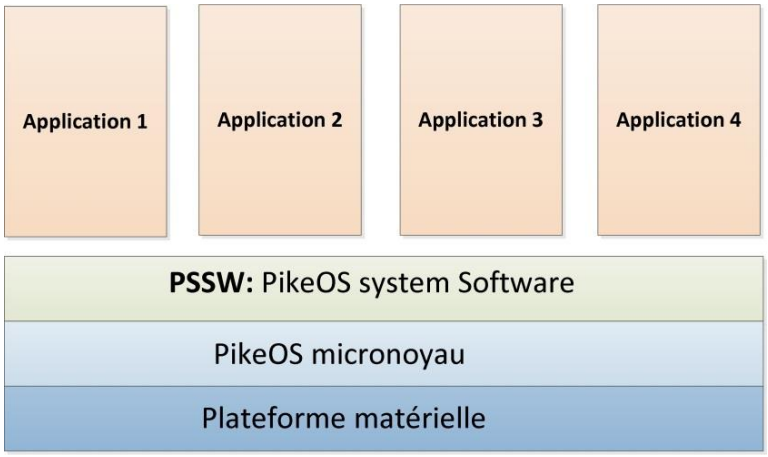

Figure 2-3: l'Architecture PikeOS

La couche micronoyau de PikeOS traite principalement quatre entités de base: la partition de ressources, la partition de temps, la tâche (équivalent au processus) et le fil d'exécution (thread).

Il fournit ensuite des services à ces entités: la préemption pour multitâche, la priorité, l'ordonnancement, la gestion de temps et des ressources. D'autre part, l'API PSSW qui se trouve au-dessus de la couche micronoyau, propose des services traditionnels : la configuration de système, la gestion des partitions, le partitionnement temporel, les services de fichiers, le contrôle de processus, la communication inter-partition et la gestion de panne. La couche micronoyau et l'API PSSW jouent le rôle de MOS pour un système IMA.

Plusieurs personnalités sont supportées par la partition de PikeOS [16], ces « personnalités » jouent le rôle de POS (OS de Partition). La personnalité peut être un OS tel que Linux embarqué, elle peut être un RTE (Run-Time Environment) tel que le POSIX, elle peut aussi être un API tel que l'ARINC653. Théoriquement, il est possible de créer un système très complexe se composant de plusieurs partitions, chaque partition ayant une personnalité différente. Ces différentes personnalités fonctionnent donc concurremment sur le même micronoyau de PikeOS. En ce qui concerne l'industrie avionique, PikeOS peut intégrer des applications avioniques créées par différents fournisseurs sur le même module IMA.

2.1.3.2 Environnement de développement

PikeOS a son propre environnement de développement: CODEO [36], une plateforme IDE basée sur Eclipse. Dans la section 4.4.1, la figure 4-6 présente un exemple de projet pour cette plateforme.

CODEO vise à mettre ensemble sur la même plateforme tous les matériaux nécessaires pour le développement des applications, le développement de noyau de PikeOS, la configuration de systèmes et la génération de solution ROM pour une cible embarquée de PikeOS.

Le développement des applications consiste à créer un projet de type « Application Developer ». Pour avoir les services APEX, il faut inclure l'API ARINC653 qui est défini dans la librairie de la plateforme. Parallèlement, pour configurer le système, il faut créer un projet de type « System Integrator » permettant de planifier et spécifier la configuration de partitions en utilisant un éditeur graphique. Les informations de configuration (mémoire, ordonnancement, entrée/sortie, etc.) sont ensuite générées automatiquement en fichier XML. La dernière étape de développement consiste à assembler tous les composants logiciels (eg. les fichiers en-tête, le BSP, le micronoyau, les applications d'usager, la configuration de système, etc.) et les générer en fichier ROM. Ce dernier va être finalement téléchargé dans la cible.

2.1.4 POK

2.1.4.1 Description

POK (PolyORB Kernel) [23] a été conçu par une équipe de recherche française dans le but d'expérimenter les architectures partitionnées et de développer des systèmes embarqués temps réel. Comme les autres OS partitionnés, il se concentre sur la sûreté et la sécurité en isolant les partitions en termes de temps et d'espace.

POK est conforme à plusieurs standards de l'industrie, y compris la première partie du standard ARINC653. Il supporte donc des fonctionnalités de l'API APEX. POK reste actuellement un prototype, il peut être utilisé pour faire une exploration de base de l'architecture IMA pour des usages académiques.

2.1.4.2 Environnement de développement

Par rapport aux autres environnements d'exécution ARINC653, un point intéressant est que POK est intégré à un flot de conception [39], tel qu'illustré à la figure 2-4. À partir d'un modèle AADL (présenté en 2.2.1), la configuration des systèmes IMA sous POK et une partie du code de ses applications peuvent être automatiquement générées.

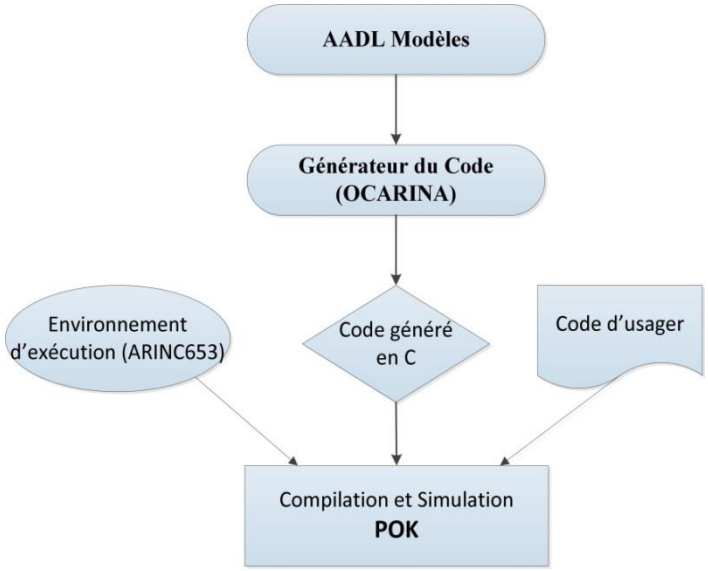

Figure 2-4: Flot de conception en AADL

2.2 Langage de modélisation

2.2.1 AADL

L'ingénierie dirigée par les modèles (MDE de l'anglais Model Driven Engineering) est considérée dans la littérature comme une solution intéressante pour le développement des systèmes IMA [17]. Le langage de modélisation AADL (Architecture Analysis and Design Langage) propose une méthodologie pour modéliser des systèmes embarqués complexes. C'est un langage de type ADL (Architecture Description Language) standardisé par la société SAE internationale, une communauté active dans le domaine de l'aérospatiale.

Par rapport aux autres langages de modélisation, un avantage évident d'AADL est qu'il regroupe une suite d'outils avec lesquels il est possible de modéliser à la fois le matériel et le logiciel d'un système complexe. Cet ensemble d'outils permet de décrire l'architecture d'un système IMA par des composants et leurs interconnexions [18]. Dans les librairies du langage AADL, des composants nécessaires pour construire des systèmes embarqués complets sont disponibles. Ces composants sont regroupés principalement en trois catégories [17]: les composants logiciels (programme, sous-programme, données processus, etc.), les composants matériels qui sont en fait des éléments pour décrire la plateforme d'exécution (processeur, mémoire, bus, processeur virtuel, bus virtuel, etc.), et les composants hybrides constitués d'une combinaison de matériel et de logiciel. Voici quelques exemples d'architectures utilisant ces différents composants: un sous-programme modélisant un code d'application, un processus modélisant un morceau de mémoire continue contenant des séquences d'exécution (threads), un processeur modélisant un microprocesseur avec un OS minimal (e.g., simplement un ordonnanceur), une mémoire modélisant un disque dur, etc. Les entités qui représentent ces composants sont représentées par des propriétés. Pour décrire un processus, on utilisera une liste de propriétés avec des mots-clés tels que période, priorité et échéance.

AADL est un langage extensible qui permet de définir de nouvelles propriétés et de créer de nouvelles annexes selon les besoins. À titre d'exemple, en 2011 AADL a été étendu pour supporter l'ARINC653 (voir AADL 2.0 [19]). Il est donc possible d'utiliser le langage AADL afin de modéliser l'architecture IMA tout en respectant les définitions du standard ARINC653. AADL a déjà été impliqué dans certains projets pour modéliser, vérifier et implémenter des

systèmes critiques, tels que le projet Flex-eZare [20] et le projet SAVI [21]. Pour modéliser une architecture définie par l'ARINC653, il faut bien représenter les entités de l'architecture IMA par les composants correspondantes qui existent dans les librairies. Avec la nouvelle version AADL 2.0, il est donc possible de décrire les partitions, les processus ARINC653, la communication intra-partition, la communication inter-partition et la gestion de panne.

Un second avantage d'utiliser AADL pour créer des systèmes avioniques est qu'une suite d'outils d'aide à la conception utilisant en entrée AADL est disponible à titre de logiciel libre. En effet, OCARINA [40] est un générateur de code pour AADL. Il s'agit d'un outil indépendant. Il fournit un ensemble de bibliothèques pour la manipulation de modèles AADL: analyse syntaxique, sémantique, exploration de modèle. Partant de ces constructions, il est possible de construire des outils évolués d'analyse tels que Cheddar lui-même, ou des générateurs de code. OCARINA est capable de générer un code C servant à initialiser le système et à instancier des composants à l'aide des fichiers de configuration XML. Tous ces fichiers et codes générés permettent l'utilisation du système d'exploitation POK. La figure 2-4 illustre le flot de conception avec AADL et le générateur de code OCARINA.

2.2.2 Simulink

Simulink est un logiciel de modélisation de systèmes dynamiques et de simulation multidomaine développé par la compagnie *MathWorks* [22]. Il s'agit d'un outil intégré dans une plateforme plus importante et très populaire – *MATLAB*. Par rapport au langage AADL, qui décrit un système avec du code, Simulink propose un environnement de développement graphique qui augmente l'efficacité de développement. Un ensemble de bibliothèques de différents domaines d'application permet le design précis, la simulation, l'implémentation et le contrôle de systèmes de communications et de traitement du signal.

L'outil Simulink permet de modéliser une fonctionnalité ou un système, simuler son comportement et vérifier les résultats avant son implémentation. Avec Simulink, il est possible de créer un système avec des composants numériques et/ou analogiques. Des modifications peuvent être facilement apportées à tout moment en changeant certains éléments du diagramme de blocs pour assurer que les spécifications soient bien respectées. Dans le cadre de ce projet, nous utiliserons une libraire existante de composants aérospatiaux (Figure 2-5). Cette librairie permet de créer des systèmes avioniques ou de simuler certaines fonctionnalités d'un système avionique,

comme par exemple le système de propulsion, l'environnement de vol et les actuateurs avioniques. Une fois le diagramme bloc bien connecté, il faut le compiler et l'exécuter. Le processus de compilation contient l'étape de vérification. Durant la compilation du modèle, la vérification de syntaxe et sémantique est faite pour assurer que le modèle soit bien complété [22].

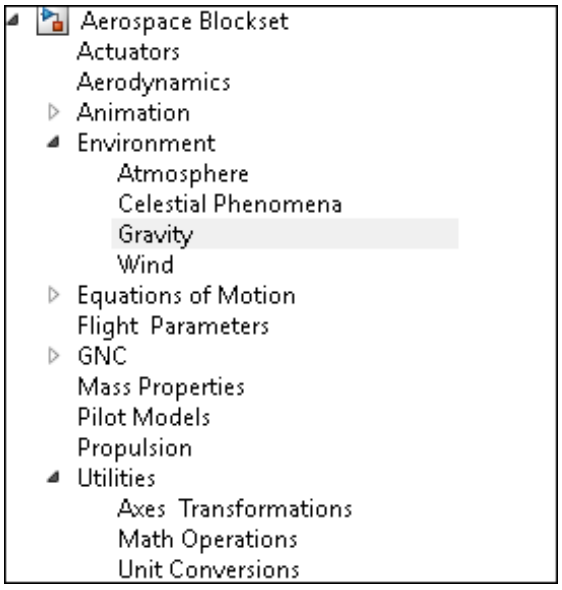

Figure 2-5: blocs aérospatiaux de Simulink

Un compilateur de modèle est aussi disponible avec Simulink. Après avoir validé le modèle, ce compilateur permet la génération de code (e.g., C/C++) à partir du modèle pour un certain nombre de cibles matérielles ou plateformes [24]. Par exemple, il permet de générer du code pour du temps réel en général ou encore pour une cible du domaine automobile tel que l'AUTOSAR[3].

Le flot de conception en Simulink est illustré à la figure 2-6. Dans cet exemple, le code généré est destiné à la plateforme x86.

[3] AUTOSAR : c'est une architecture ouverte et standardisée conçue par les fabricants, les fournisseurs et les développeurs des outils du domaine automobile.

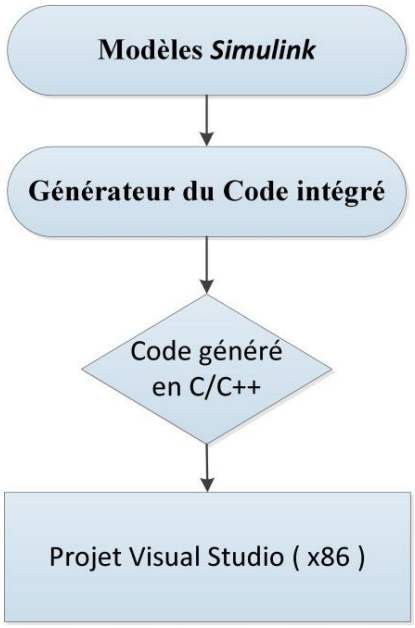

Figure 2-6: Flot de conception avec Simulink

2.3 Bus de communication

2.3.1 Le protocole TCP/IP

Le protocole TCP et le protocole IP sont deux protocoles de communication de réseaux informatiques très importants dans le modèle TCP/IP illustré par la figure 2-7.

D'après la définition du standard OSI (Open System Interconnexion), qui est un standard de communication de systèmes informatiques, l'architecture du modèle OSI est composée de sept couches: application, présentation, session, transport, réseau, liaison de données et physique.

Le modèle TCP/IP peut être considéré comme une version simplifiée du modèle ISO. Il se compose de 4 couches : la couche application, transport, réseau et accès réseau. Il existe une relation de correspondance entre ces deux modèles [25] (voir la figure 2-7).

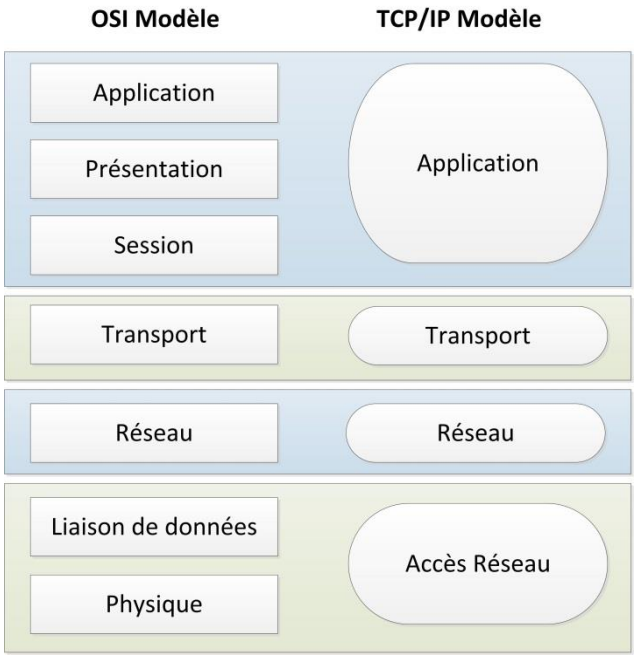

Figure 2-7: Modèle OSI vs modèle TCP/IP

Chaque couche du modèle est définie pour résoudre un certain nombre de problèmes. TCP est situé au niveau de la couche transport pour proposer le service de découpage du flux de données en segments. IP est situé sur la couche réseau, il fournit le service d'adressage pour les ordinateurs qui se communiquent.

Le protocole TCP/IP est souvent utilisé aussi pour le transfert de données entre les systèmes embarqués. Il ne peut pas être utilisé directement dans les systèmes avioniques à cause des limitations au niveau fiabilité, vitesse, etc. Toutefois, dans un contexte de simulation haut niveau de système avionique, il peut être intéressant de le considérer comme une abstraction d'un bus.

2.3.2 AFDX et ARINC664

Le bus AFDX (Avionics Full Duplex Switched Ethernet) a été développé et standardisé par les industriels européens de l'avionique. Il est basé sur le réseau Ethernet et respecte la spécification du standard ARINC664 (section 7) [7]. Il offre une bonne fiabilité pour l'échange de données entre les divers composants d'un système avionique commercial.

Deux autres standards de communications sont aussi utilisés dans le domaine avionique. Il s'agit d'ARINC429 et de MIL-STD 1553B [25]. Ces deux réseaux sont semi-duplex. D'après la définition du standard ARINC429, les différentes unités sont connectées point à point via un lien

direct. Le standard MIL-STD 1553B définit un bus commun pour connecter ensemble plusieurs unités. Par rapport au réseau AFDX, la performance de ces deux types de bus a plusieurs limitations dont celle de la bande passante. Ils contribuent également à augmenter considérablement le poids de l'avion, ce que les fabricants d'avions essayent aujourd'hui de minimiser.

Le réseau avionique AFDX vise donc à résoudre ces problèmes. Il remplace les connexions point à point utilisées dans les systèmes distribués par la connexion virtuelle [25] (en anglais « Virtual Links »). Par définition, des chemins de données peuvent être créés par le logiciel entre les différentes unités, dans le but d'établir des connexions virtuelles actives sur un réseau physique intégré, soit le réseau AFDX. Ici, la caractéristique « active » veut dire qu'en cas de défaillance d'une connexion, la reconfiguration d'une nouvelle connexion peut se faire rapidement par logiciel. Grâce à la caractéristique « intégré », il est théoriquement possible d'établir rapidement une connexion virtuelle entre n'importe quelle paire de modules avioniques par logiciel. Les formats de données envoyées et reçues doivent être strictement identiques. Le réseau AFDX est déterministe [25], autrement dit, le temps de propagation sur le réseau est connu et garanti.

AFDX est considéré sans doute comme une nouvelle génération de réseau avionique. Ce type de réseau a plusieurs avantages. Premièrement, il fait diminuer le poids et la dimension du câble, tout en augmentant la bande passante. De plus, ce réseau est extensible, de sorte qu'un nouveau module pourrait être intégré rapidement dans le réseau. Sa mise à jour est donc simple. Finalement, le concept derrière AFDX se marie parfaitement avec le concept de plateforme IMA vu à la section 2.1. Ce mariage de deux nouveaux concepts permet donc la réalisation de systèmes avioniques beaucoup performants.

2.4 Analyse basée sur la revue de littérature

Tel que mentionné dans la section 1.4, les travaux de recherche du présent mémoire consistent à poursuivre la recherche faite par un autre membre du projet J. Savard [8]. Ses travaux se sont concentrés sur l'étude et l'identification des particularités des différents environnements d'exécution conformes à la norme ARINC653 dans le but de choisir le plus adapté. À part POK et SIMA, nous avons aussi étudié deux autres environnements d'exécution : AIR [42] et XtratuM [24]. AIR est un RTOS partitionné conforme à la norme ARINC653 et il est aussi une cible de

l'outil de configuration CONFIGUIMA. XtratuM est un moniteur de machines virtuelles. Il est aussi une des cibles du générateur de code OCARINA.

Ces logiciels fournissent tous un environnement d'exécution ARINC653. Dans les travaux de M. J. Savard, le coût d'une licence, la disponibilité du support, la conformité aux normes ARINC653, la disponibilité à une bonne documentation et l'intégration à un flot de conception dirigé par les modèles ont été pris en compte comme des facteurs de sélection.

Une comparaison a été faite entre ces 4 environnements d'exécution (XtratuM, AIR, POK et SIMA): XtratuM, AIR et POK sont dépendants de la plateforme matérielle cible. Comme beaucoup de logiciels libres, ils souffrent d'un manque de documentation. XtratuM et POK ne sont pas entièrement conformes au standard ARINC653. Bien que le simulateur SIMA puisse offrir une bonne précision sans besoin d'avoir recours à la plateforme matérielle, ni d'avoir besoin d'un vrai RTOS supportant le standard ARINC653, il n'offre pas d'outils de développement comme POK. Tel que mentionné plus tôt, POK est intégré dans un flot de conception AADL. Plus précisément, l'environnement OCARINA permet une génération automatique d'une description AADL vers POK. Mais le principal désavantage de POK est son trop faible degré de développement qui est encore au niveau de prototype. En résumé, nous sommes arrivés à la conclusion qu'au moment de débuter le projet, aucun environnement d'exécution ARINC653 ne rencontrait toutes les conditions de sélection mentionnées ci-dessus.

Dans de vrais systèmes avioniques complexes, les capteurs et actuateurs sont de vrais composants physiques. En phase de développement des systèmes IMA, pour faire la validation fonctionnelle, les applications avioniques doivent souvent échanger des données ou des signaux avec les capteurs et actuateurs. Il faut savoir que ces composants peuvent être très dispendieux. De plus, le bon fonctionnement de ces composants physiques dépend parfois du bon environnement de vol (eg. la vitesse, l'altitude, etc.). Pour résoudre ce problème, on utilise souvent des logiciels pour simuler le comportement des capteurs et actuateurs afin de compléter la simulation d'un système complet. Dans ce mémoire, on cherche à identifier un outil peu coûteux permettant de développer efficacement les logiciels visant la simulation des capteurs et actuateurs avioniques.

Dans un système avionique typique, le module IMA communique avec les capteurs et actuateurs en utilisant le bus de communication AFDX. Quand les composants en logiciel sont utilisés pour

remplacer ces capteurs et actuateurs en matériel, on a besoin d'un bus de communication peu coûteux pour échanger les données.

Cette analyse basée sur la revue de littérature confirme donc la problématique de ce mémoire : on cherche une méthodologie efficace en termes de temps et coût pour développer et simuler les systèmes IMA. Nous cherchons un environnement d'exécution qui est à la fois peu coûteux, entièrement conforme à la norme ARINC653, indépendant de plateforme matérielle et intégré dans un flot de conception MDE, et offrant un support technique et une documentation. Nous avons également besoin d'un outil peu coûteux pour développer efficacement les capteurs et actuateurs avioniques. Nous cherchons aussi un moyen pour remplacer le bus de communication AFDX qui coûte très cher.

CHAPITRE 3 PROPOSITION D'UN FLOT DE CONCEPTION POUR PLATE-FORME IMA

Suite à la problématique soulevée au chapitre 1, ce chapitre propose un ensemble de solutions basées sur la revue de littérature présentée au chapitre 2. Dans ce chapitre, un flot de conception est présenté. Ce flot a pour but de concevoir efficacement des systèmes avioniques et d'en faire une cosimulation. Après avoir complété la vérification du comportement, un processus pour réaliser l'implémentation est proposé, ce qui implique de porter l'application et la configuration du système vers un environnement d'exécution avionique.

3.1 Présentation générale du flot

Une image globale du flot réalisé dans ce travail est présentée à la figure 3-1. Ce flot est lui-même composé de 3 niveaux : 1) niveau de modélisation, 2) niveau de cosimulation et 3) niveau d'implémentation. C'est un ensemble de solutions visant donc à modéliser, simuler et implémenter des systèmes avioniques rapidement.

Tout en haut, le niveau de modélisation contient les deux flots de conception dirigés par les modèles AADL et Simulink. Tel que mentionné précédemment, le flot en AADL sert à intégrer efficacement l'API ARINC653 et d'en faire la configuration pour le simulateur SIMA (le choix de SIMA est justifié à la section suivante). Ce passage d'AADL à SIMA est assuré par OCARINA. Le flot de conception de Simulink cherche à développer rapidement des périphériques avioniques tels que les capteurs et actuateurs pour simuler l'environnement de vol.

Au deuxième niveau, la plateforme de cosimulation, est composée du module IMA simulé par SIMA et des périphériques avioniques simulés par Simulink. L'échange de données entre ces deux parties est fait via le réseau internet TCP/IP.

Finalement, au troisième niveau, nous avons choisi le logiciel PikeOS comme composante maîtresse de la plateforme d'implémentation qui se trouve au dernier niveau. À ce niveau, on reprend directement les applications avioniques et la configuration pour les systèmes de SIMA. Un serveur/adaptateur suivant le protocole TCP/IP est utilisé pour faire communiquer SIMA avec les capteurs et les actuateurs logiciels.

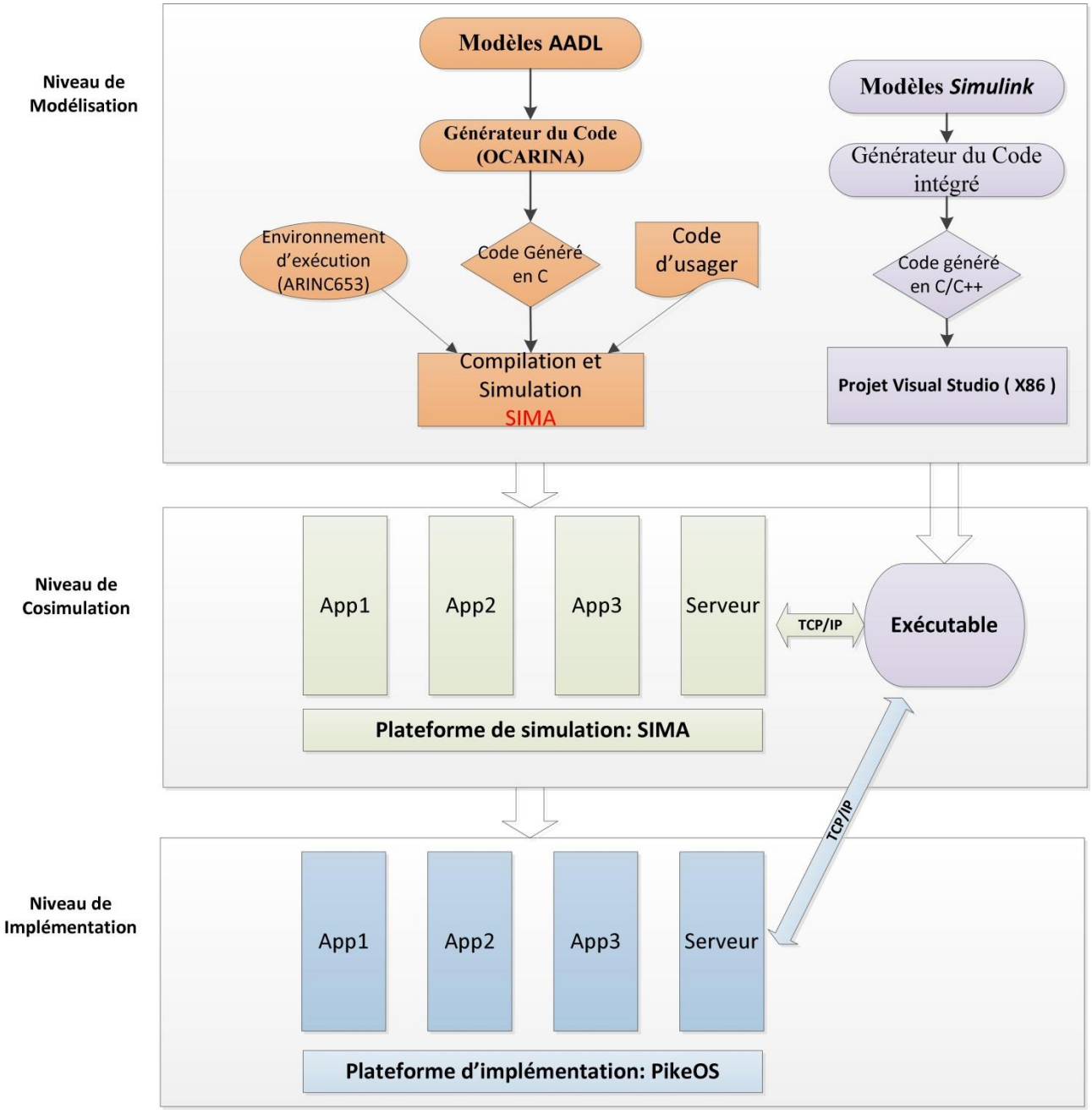

Figure 3-1: Flot général de conception

3.2 Niveau de modélisation

3.2.1 Niveau AADL

Tel qu'expliqué à la section 2.2.1, le langage AADL supporte plusieurs concepts pour modéliser des systèmes avioniques. Il est capable de modéliser à la fois le logiciel et le matériel pour les

systèmes embarqués. Il offre aussi une suite d'outils pour faire l'exploitation plus approfondie des modèles dont OCARINA destiné à générer du code pour l'OS partitionné POK.

D'autre part, il existe plusieurs environnements d'exécution ARINC653. Dans la section 2.4, nous avons réalisé une comparaison entre ces environnements, soit AIR, XtratuM, POK et SIMA. Cette comparaison constitue en fait une étude de faisabilité que nous présentons à la section 3.2.2.1. Notre choix s'étant arrêté à SIMA.

La figure 3-2 montre le flot de conception résultant.

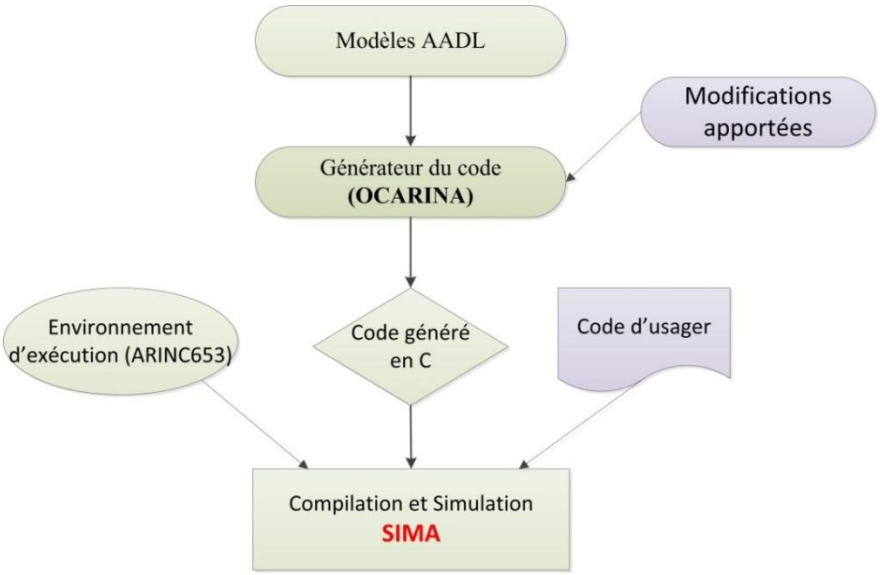

Figure 3-2: Flot de conception pour SIMA

Après avoir présenté les résultats de l'étude de faisabilité, la section suivante explique en détail les modifications apportées pour concevoir ce nouveau flot, puis dans le chapitre 4, une étude de cas sera présentée afin d'en faire la validation.

3.2.2 Mise en œuvre

3.2.2.1 Étude de faisabilité

Bien que l'auteur du présent mémoire ait contribué à cette étude, son principal maître d'œuvre est J. Savard [8]. Dans ce qui suit nous présentons un résumé de cette étude, aussi disponible dans [29].

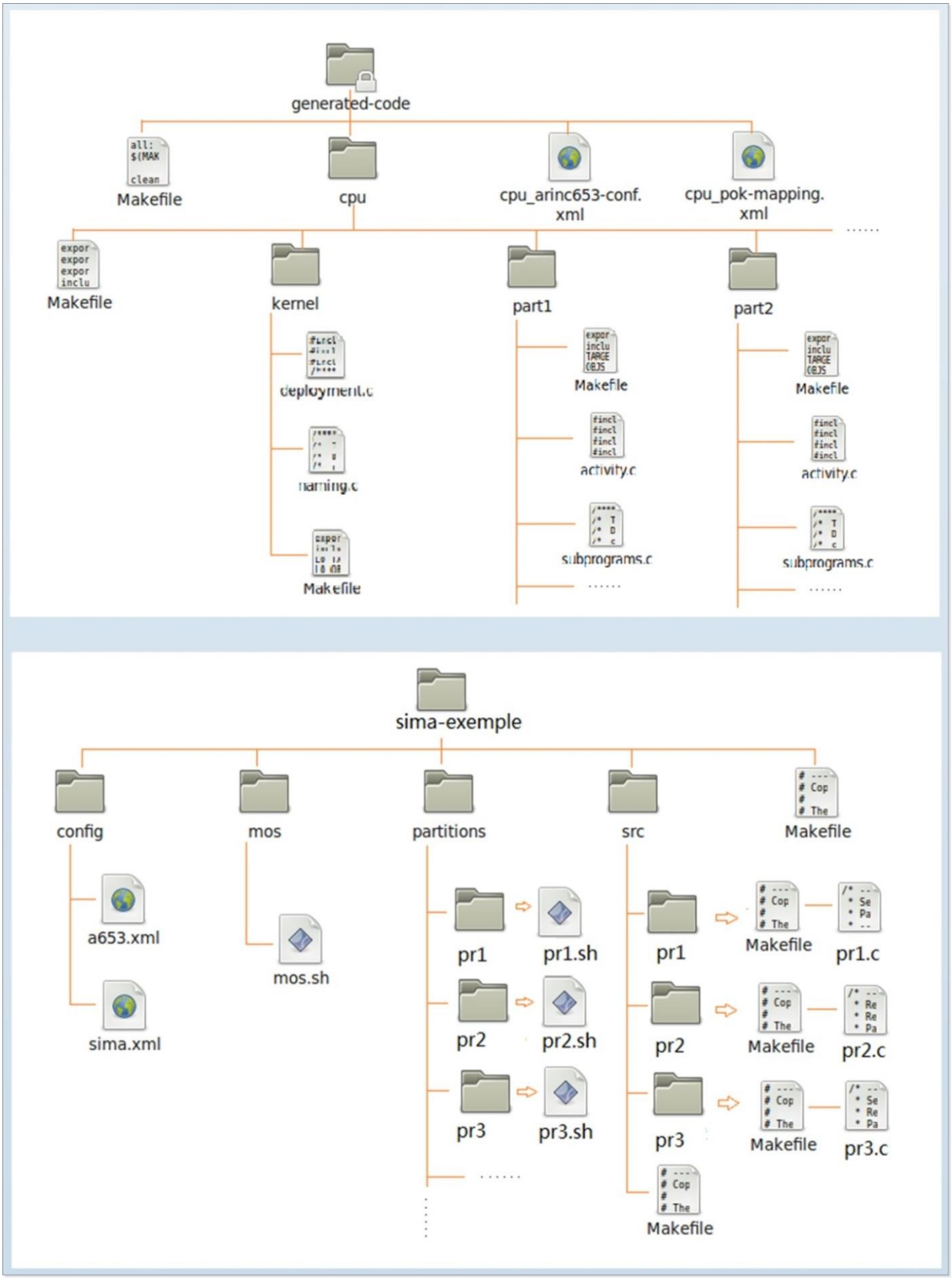

Figure 3-3: Comparaison de structures d'applications pour POK(en haut) et SIMA

Tout d'abord, d'après la comparaison faite à la section 2.4, nous avons présenté une proposition de flot de conception basé sur une approche de MDE qui comporte un environnement d'exécution ARINC653 à faible coût. Le flot de conception AADL de POK a été modifié pour cibler SIMA (Figure 2-4 et 3-2).

Ensuite, l'effort a été mis à la comparaison du code source et de la configuration système pour ARINC653, sous les cibles SIMA et POK. Sur la figure 3-3, les structures d'application pour POK et SIMA sont comparées. Une application complète est composée du code d'usager, les fichiers de compilation, les fichiers de configuration en XML et les fichiers du code script dans le cas de SIMA. Des éléments de configuration unique à SIMA qui le distinguent de POK ont été identifiés. Des preuves de concept ont été faites dans le but de prouver que les fichiers contenant du code source et de la configuration système sous POK fonctionnaient aussi avec SIMA. La réussite de ces preuves de concept (figure 3-4) confirmait donc que l'idée de cibler SIMA à partir d'OCARINA était intéressante pour réaliser une première simulation du système avant même de procéder à son implémentation.

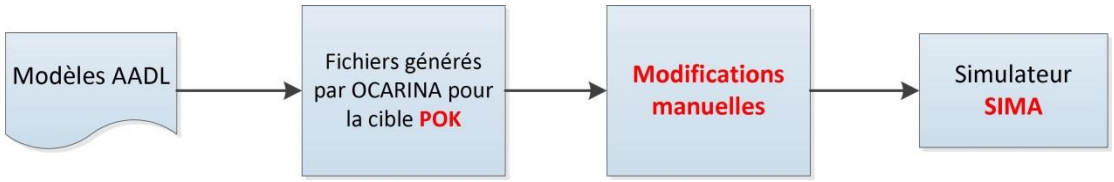

Figure 3-4: Preuve de concept

En fin de compte, SIMA est une solution intéressante qui correspond à nos besoins (e.g., support ARINC653 complet, simulation à haut niveau offrant toutefois un bon degré de précision et faible coût pour acquérir une licence). Toutefois, SIMA n'est actuellement pas supporté par OCARINA (qui lui supporte AADL tout en étant du domaine public). Nous nous sommes donc attaqués à faire cette intégration. La figure 3-5 explique le flot de travail pour réaliser cette intégration et ainsi permettre un passage automatique d'une description AADL afin de configurer et préparer les données pour une simulation sous SIMA. Dans ce flot de travail, des modifications sont apportées sur le code source de l'OCARINA pour qu'il puisse directement générer une application vers le simulateur SIMA.

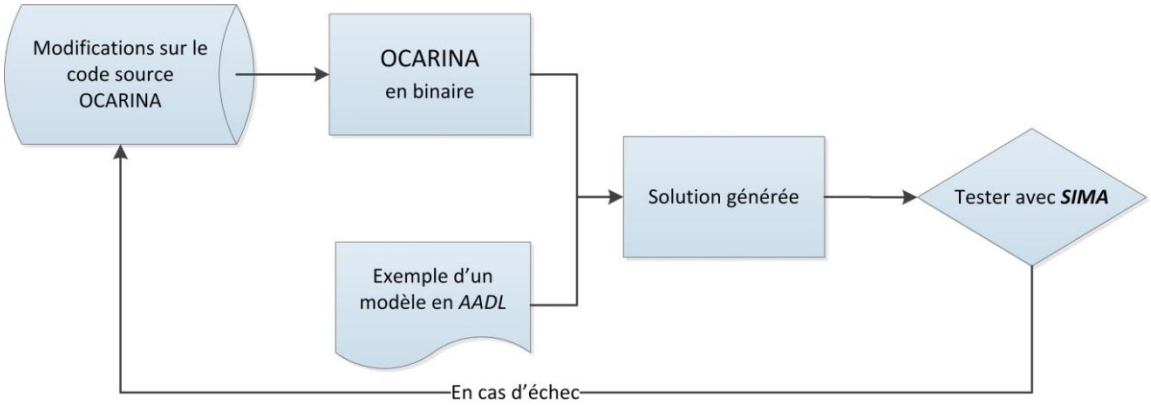

Figure 3-5: Flot de travail

3.2.2.2 Automatisation du flot

Une description en AADL est constituée d'une série de déclarations (voir Annexe 5). Ces déclarations peuvent établir une représentation virtuelle pour une architecture telle qu'IMA. Quand ces déclarations sont traitées par OCARINA, il faut les instancier avant de pouvoir évaluer l'architecture [28]. Donc, à partir d'une description AADL, le processus de traitement par OCARINA se décompose en deux étapes: 1) il faut d'abord créer une représentation (structure de données) du modèle AADL, et 2) il faut instancier cette représentation pour ainsi vérifier et valider l'architecture.

À l'intérieur du noyau d'OCARINA, la solution à générer est représentée par un « arbre » qui est basé sur des « nœuds » représentant des fichiers, des fonctions, des processus, des propriétés, etc. Les valeurs déclarées dans AADL sont utilisées pour instancier les paramètres de ces nœuds (e.g. échéance, période, etc.). Tel qu'illustré à la figure 3-6, cet « arbre » est composé de nombreux nœuds. Un de ceux-ci représente la racine de cet arbre. Un nouveau nœud peut être inséré à un nœud déjà existant. Ce dernier s'appelle donc le « nœud-parent », et parallèlement, celui inséré est appelé le « nœud-fils ». En résumé, quand on désire créer un nouveau nœud, il faut appeler la fonction *Add_New_Node* (définie dans la librairie d'OCARINA), en spécifiant certains paramètres comme la position du nœud, son nom et son nœud-parent. Cette opération permet donc de définir un nouveau nœud et de l'insérer automatiquement comme nœud-fille sur un nœud-parent.

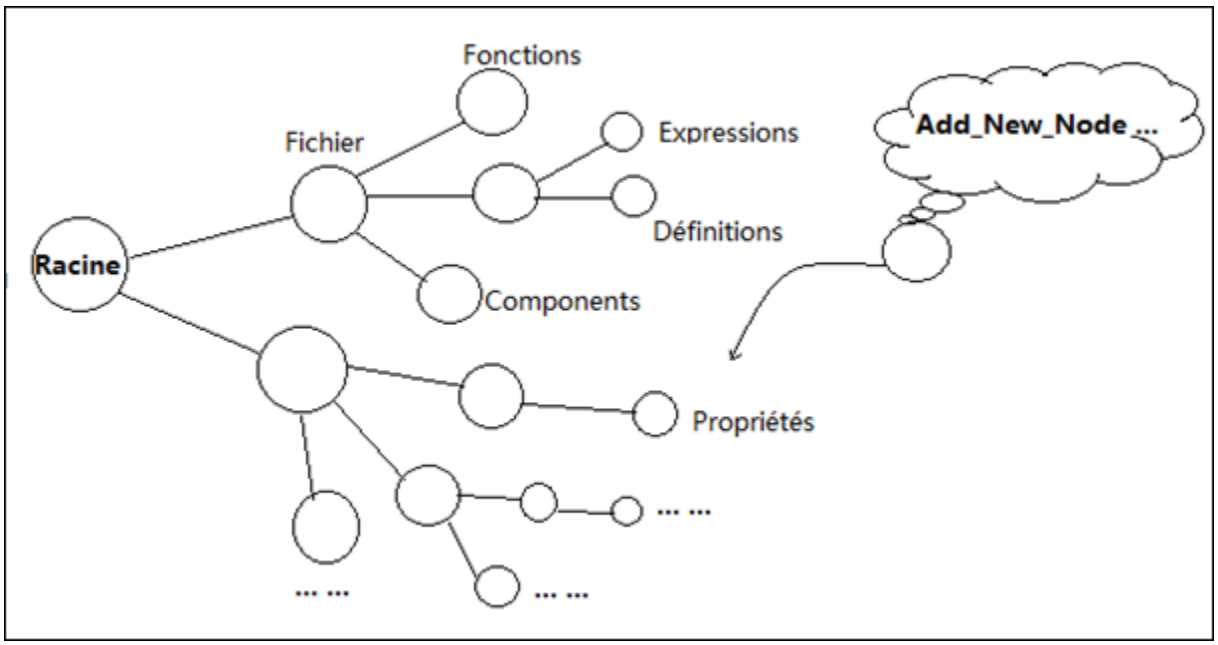

Figure 3-6 : Structure interne d'OCARINA sous forme d'arbre

Par défaut, la représentation existante sous OCARINA est celle ciblant POK (puisque ce dernier est supporté par OCARINA). Nos travaux ont donc consisté à modifier cette représentation pour qu'elle puisse s'adapter au simulateur SIMA.

Tel qu'illustré à la figure 3-4, l'ensemble des fichiers qui forme une application fonctionnant sous le simulateur SIMA est composé de code C effectuant des appels aux fonctionnalités APEX, de deux fichiers de configuration XML du système, de fichiers de compilation (ou soit « *Makefile* ») et de scripts de lancement des applications. Tous ces fichiers se retrouvent sous le même répertoire. Suite à la l'étude de faisabilité effectuée en [8], certains fichiers générés pour POK peuvent être réutilisés en totalité, alors que certains peuvent être réutilisés en partie seulement. Les sections suivantes expliquent plus en détail les travaux de modifications apportées.

3.2.2.3 Réorganisation des fichiers pour une génération SIMA à partir de AADL

1. **Gabarit pour la génération du code C (fichiers .c)**

On s'intéresse ici à la génération de fichiers .c (Figure 3-4). En général, le travail de modification se trouve au niveau du code effectuant la création de processus, de fil d'exécution ou de l'appel d'une fonction APEX, puisque c'est à ce niveau qu'OCARINA opère la génération automatisée. Un exemple de génération est donné à la figure 3-7 (ou à l'Annexe 1). En résumé, pour permettre

une simulation sous SIMA, il a fallu modifier OCARINA afin qu'il puisse générer un ensemble de fichiers respectant un certain *gabarit* (de l'anglais *template*). Ce dernier est principalement composé de quatre parties : 1) instructions pour inclure des fichiers en-tête (*include*), 2) déclarations de variables, 3) tableaux et d'expressions et 4) création et appels de fonctions. C'est à l'intérieur de ce gabarit compatible SIMA que sera finalement intégré le code applicatif de l'usager décrit sous AADL.

Les explications suivantes précisent les détails des modifications apportées à OCARINA pour créer ce gabarit. Notez que le nombre de fichiers à inclure, le nombre de variables, le nombre de tableaux et d'expressions, ainsi que le nombre d'appels de fonctions représentant le corps des fichiers C, est dépendant d'une application donnée. Puisque le logiciel OCARINA est conçu en langage Ada, la programmation du gabarit a été réalisée avec ce dernier.

Dans ce qui suit, nous décrivons plus en détail chaque partie du gabarit.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <a653.h>
4  #include "activity.h"
5  #include "deployment.h"
6  #include "pok_wrapper.h"
7
8  PROCESS_ID_TYPE arinc_threads[POK_CONFIG_NB_THREADS];
9  QUEUING_PORT_ID_TYPE envi_pdatain_envi_id;
10 QUEUING_PORT_ID_TYPE envi_pdataout_envi_id;
11
12 int entry_point ()
13 {
14   PROCESS_ATTRIBUTE_TYPE tattr;
15   RETURN_CODE_TYPE ret;
16   CREATE_QUEUING_PORT ("pdatain_envi", 208 * 1, 2, DESTINATION, FIFO, &(envi_pdatain_envi_id), &(ret));
17   ASSERT_RET(ret);
18   CREATE_QUEUING_PORT ("pdataout_envi", 104 * 1, 2, SOURCE, FIFO, &(envi_pdataout_envi_id), &(ret));
19   ASSERT_RET(ret);
20   tattr.ENTRY_POINT = (int*)thread_environment_job;
21   tattr.BASE_PRIORITY = 10;
22   tattr.DEADLINE = SOFT;
23   tattr.PERIOD = 1000000000;
24   tattr.STACK_SIZE = 100;
25   tattr.TIME_CAPACITY = 990000000;
26   strcpy (tattr.NAME, "thread_environment.impl");
27   CREATE_PROCESS (&(tattr), &(arinc_threads[1]), &(ret));
28   START (arinc_threads[1], &(ret));
29   ASSERT_RET(ret);
30   SET_PARTITION_MODE (NORMAL, &(ret));
31   return (0);
32 }
```

Figure 3-7: Extrait d'un code généré dans un fichier .c

- **Génération de fichiers en-tête**

Le générateur est capable d'inclure automatiquement les fichiers en-tête (.h) lorsque les fichiers de définitions (.c) sont créés. Par convention, on utilise la directive *#include* soit avec des chevrons <> représentant un fichier en-tête du système, ou soit avec des guillemets doubles *“”* signifiant un fichier en-tête d'usager. En gros, il a fallu redéfinir une table de correspondance de fonctions et leurs fichiers en-tête (figure 3-8).

```
RE_Header_Table : constant array (RE_Id) of RH_Id
  := (
      --  Runtime functions associations
      RE_Pok_Thread_Sleep               => RH_A653,
      RE_Pok_Thread_Sleep_Until         => RH_A653,
      RE_Pok_Thread_Wait_Infinite       => RH_A653,
      RE_Pok_Thread_Create              => RH_A653,
      RE_Pok_Thread_Attr_Init           => RH_A653,
      RE_Pok_Thread_Suspend             => RH_A653,
      RE_Pok_Thread_Restart             => RH_A653,
      RE_Pok_Thread_Period              => RH_A653,
      RE_Pok_Thread_Stop                => RH_A653,
      RE_Pok_Thread_Stop_Self           => RH_A653,
      RE_Pok_Error_Handler_Create       => RH_A653,
      RE_Pok_Error_Kernel_Callback      => RH_A653,
      RE_Pok_Error_Partition_Callback   => RH_A653,
      RE_Pok_Error_Ignore               => RH_A653,
      RE_Pok_Error_Confirm              => RH_A653,
      RE_Pok_Error_Get                  => RH_A653,
      RE_Pok_Error_Handler_Worker       => RH_A653,
      RE_Entry_Point1                   => RH_A653,
      RE_Strcpy                         => RH_String,
      RE_Name                           => RH_String,
      RE_Message_Addr_Type              => RH_Null,
```

Figure 3-8: Extrait d'une table de correspondance de fonctions et leurs fichiers en-tête.

Ainsi, pour chaque fonction appelée, OCARINA pourra se référer à une table de correspondance afin d'inclure un en-tête pour un fichier C. Par exemple, dans la ligne 26 du code C de la figure 3-7, la fonction *stycpy( )* est appelée, par conséquent la table de correspondance indiquera l'ajout d'une instruction « include » pour *string.h* dans l'entête du programme.

- **Déclaration d'une variable**

Comme précisé plus tôt, la représentation AADL est sous forme d'une arborescence. Elle est basée sur des « nœuds » qui représentent des fichiers, des fonctions, des processus, des propriétés, etc. Donc, la plupart des travaux de modifications sur le modèle OCARINA consistent à rajouter ou enlever des nœuds dans l'arbre. La fonction *Append_Node_To_List (Node_Id,*

List_Id) qui est définie dans la librairie d'OCARINA permet d'insérer un « nœud » sur l' « arbre ». La figure 3-9 montre le processus pour créer une déclaration de variable. Après avoir appelé les fonctions pour déclarer un nœud de type variable et puis l'insérer dans l'arbre, le résultat se trouve à la ligne 14 du code généré (Figure 3-7).

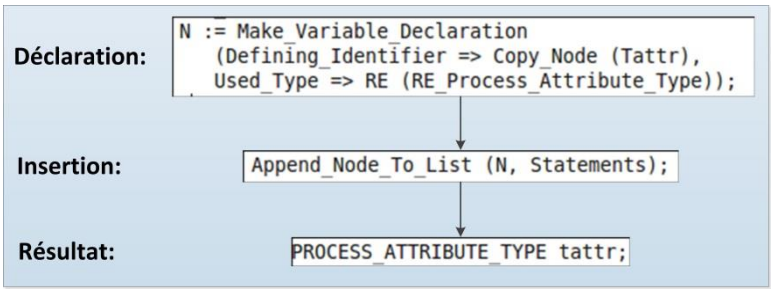

Figure 3-9: Déclaration d'une variable

- **Déclaration d'un tableau**

La syntaxe pour déclarer un tableau est similaire à celle d'une variable. Pour définir un tableau, la fonction *Make_Array_Declaration(Nom , taille)* est appelée pour nommer et dimensionner ce tableau en retournant une valeur N de type « Node_Id ». Ensuite, de la même manière la fonction *Make_Variable_Declaration* est appelée pour déclarer le type de ce tableau. Enfin, la fonction *Append_Node_To_List* va insérer ce nœud à l'endroit désiré (Figure 3-10). Le résultat se trouve à la ligne 8 du code généré (Figure 3-7).

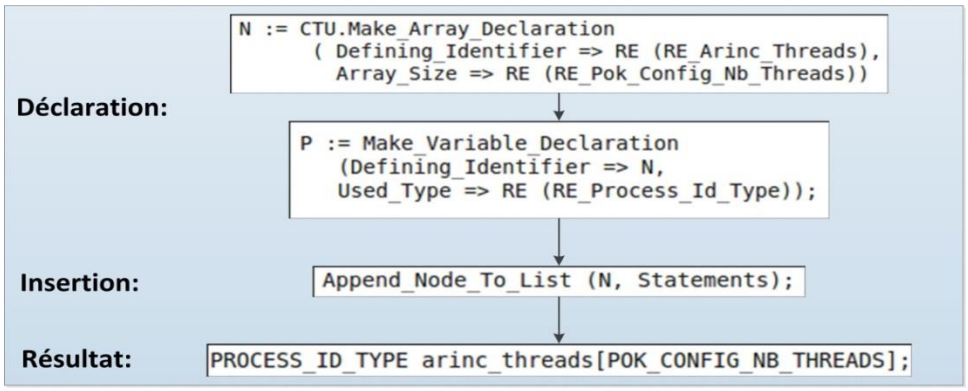

Figure 3-10: Déclaration d'un tableau

- **Déclaration d'une expression (équation)**

La figure 3-11 montre le processus pour définir une expression. Après avoir récupéré les bonnes valeurs définies dans le modèle en AADL, la fonction *Make_Expression(Expression à gauche,*

Indicateur, Expression à droite) sera utilisée pour déterminer les éléments nécessaires d'une équation. À la fin, un nœud contenant cette équation sera inséré dans l'arbre. Le résultat se trouve à la ligne 21 du code généré (Figure 3-7).

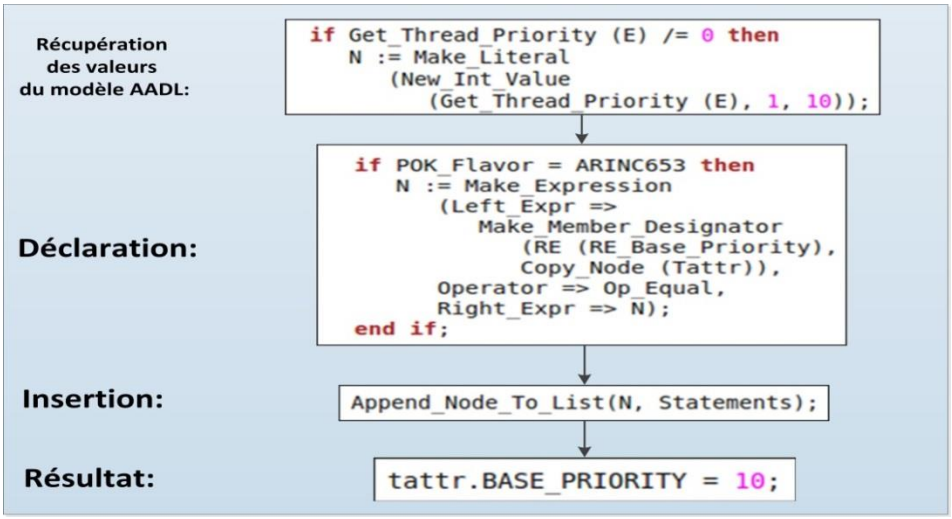

Figure 3-11: Déclaration d'une équation

- **Appel d'une fonction**

Une fonction est souvent appelée par une autre fonction. Le processus (Figure 3-12) pour réaliser cette tâche est également similaire à la déclaration d'une équation. Un certain nombre de paramètres pour la fonction appelée sont d'abord récupérés du modèle en AADL. Puis, la fonction *POK_Make_Function_Call_With_Assert (Nom de fonction, ses paramètres)* va déterminer la forme de cette fonction appelée avant d'insérer un nœud dans l'arbre. Le résultat se trouve à la ligne 27 du code généré (Figure 3-7).

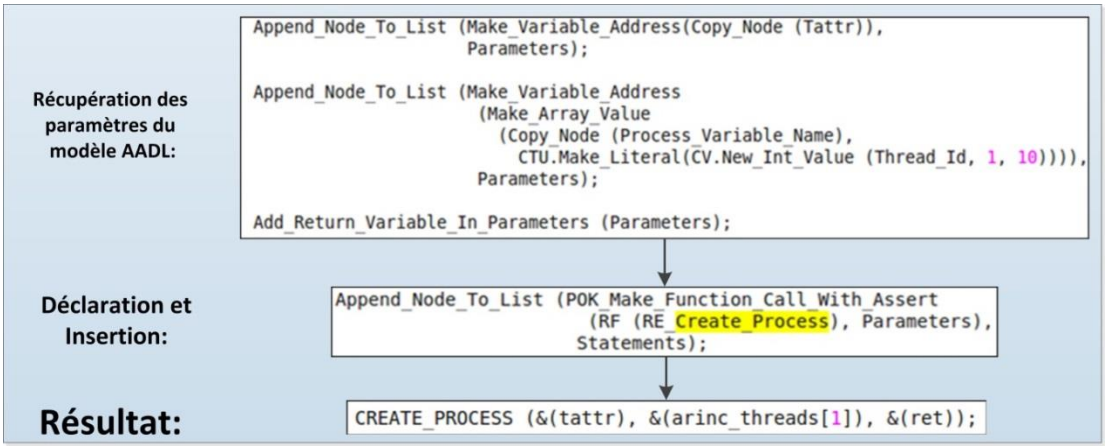

Figure 3-12: L'appel d'une fonction

- **Création d'une fonction**

La figure 3-13 montre les étapes pour créer une fonction. La fonction *Make_Function_Specification* va définir les paramètres nécessaires pour une fonction, alors que la fonction *Make_Function_Implementation* va implémenter la fonction dans un fichier également par l'insertion d'un nœud. Le résultat se trouve à la ligne 12 du code généré (Figure 3-7).

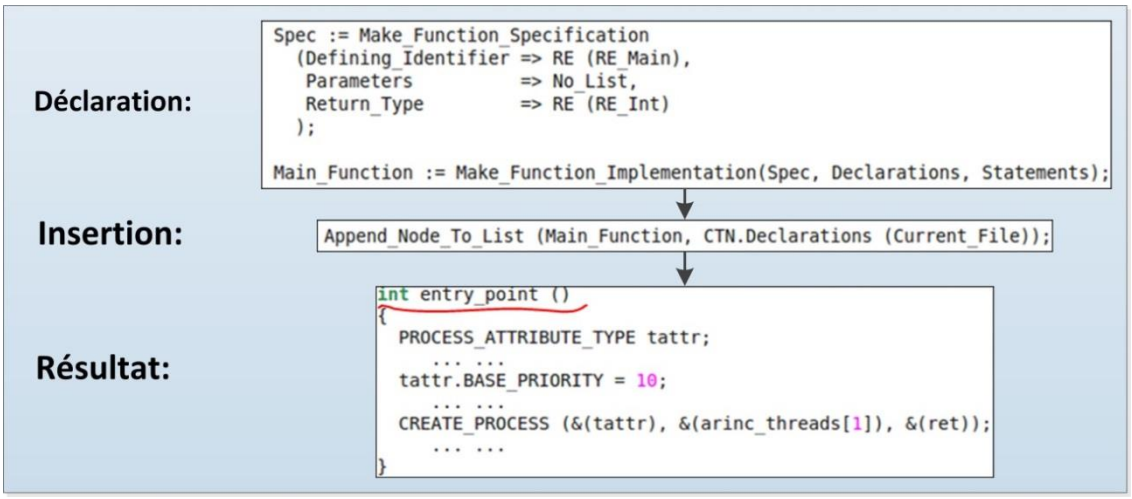

Figure 3-13: Création d'une fonction

2. **Génération des fichiers de configuration (fichiers XML)**

Deux fichiers de type XML assurent la configuration du système sous SIMA : *a653.xml* et *sima.xml* (Figure 3-4). Le premier fichier XML caractérise les éléments et les concepts propres à ARINC653 et à la gestion des erreurs. La plupart des éléments de ce fichier sont descriptibles en AADL sauf ceux qui concernent la gestion des erreurs. Le deuxième fichier XML configure des éléments propres à SIMA [8]. La plupart des éléments de ce fichier peuvent aussi être modélisés en AADL. Pour les éléments qui ne peuvent pas être décrits sous AADL (e.g., une information qui tient compte d'un certain contexte), le générateur les prend en charge[4]. L'annexe 2 illustre un exemple de ces deux fichiers générés en prenant un modèle simple en AADL (arinc653-queueing). Quelques méthodes importantes pour générer les fichiers XML sont présentées dans ce qui suit.

[4] Par exemple, AADL ne donne pas d'adresse IP. Une valeur doit être générée.

Comme montré à l'Annexe 2, dans un fichier XML, des définitions et des mots clés sont toujours encadrés par les chevrons < >. Sous le format XML, un encadré de chevrons est aussi appelé une balise, mais elle doit être fermée. Ces balises fermées contiennent les informations nécessaires concernant la gestion de panne et les concepts propres à ARINC653. La figure 3-14 explique de quelle manière un nœud représente une balise fermée.

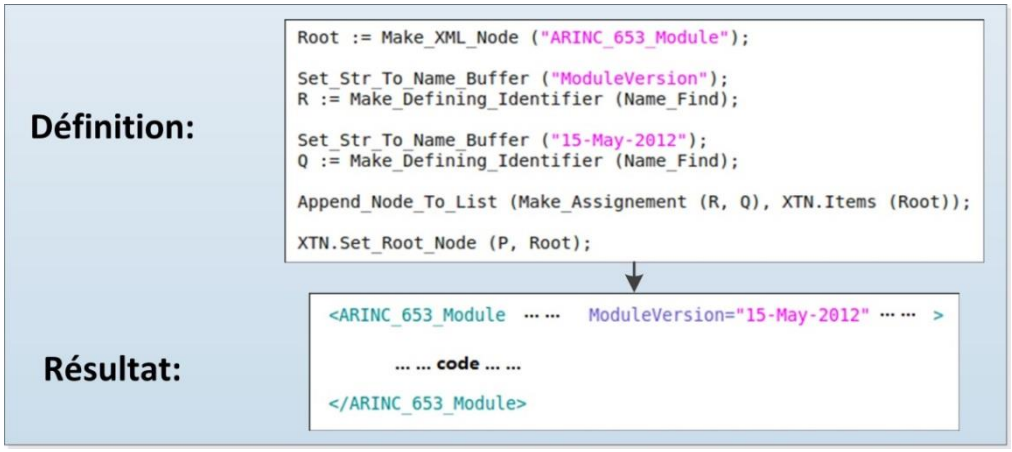

Figure 3-14: Définition de module pour fichier XML

Il arrive parfois que ces encadrés de chevrons soient eux-mêmes encapsulés dans une autre balise fermée, comme c'est le cas du résultat donné dans la figure 3-15.

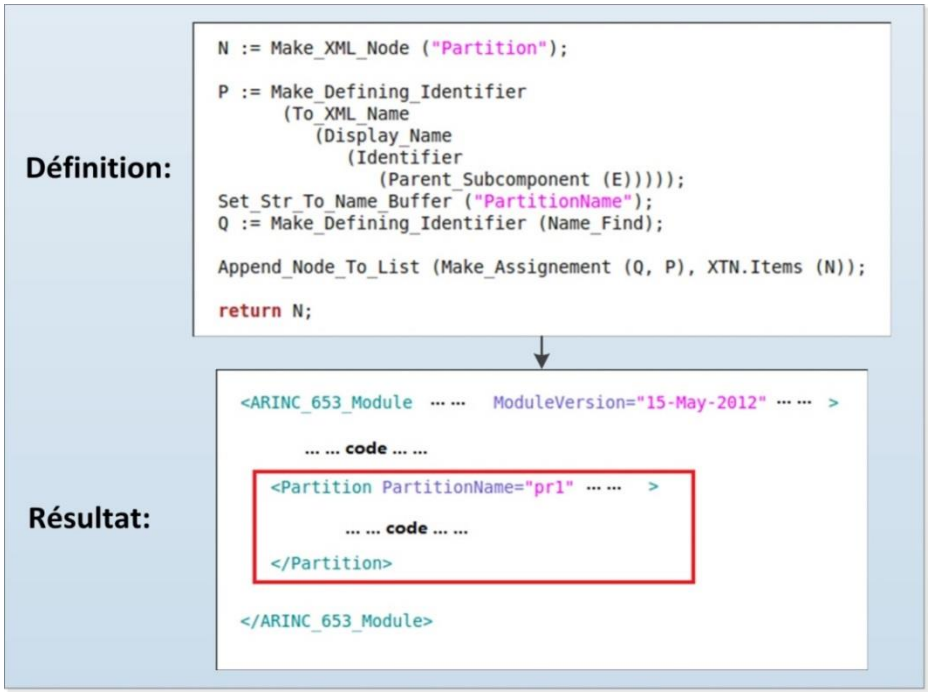

Figure 3-15: Définition de module pour fichier XML

3. Modifications apportées aux autres fichiers

Pour assurer le bon fonctionnement d'un système sous SIMA, d'autres fichiers sont aussi nécessaires, y compris les fichiers de compilation (en anglais « *Makefile* ») et les scripts de lancement de partitions (fichiers .sh) (Figure 3.4). Les fichiers de compilation et la plupart des scripts de lancement contiennent de l'information créée dynamiquement. Plus précisément, les variables ne sont pas toujours descriptibles en AADL, mais l'information peut être construite dynamiquement en référençant le contenu de certains nœuds, par exemple, un répertoire, le nom d'un répertoire, etc.

La figure 3-16 montre la méthode utilisée pour créer un fichier de compilation, en fait, similaire à la création d'un script de lancement de partitions.

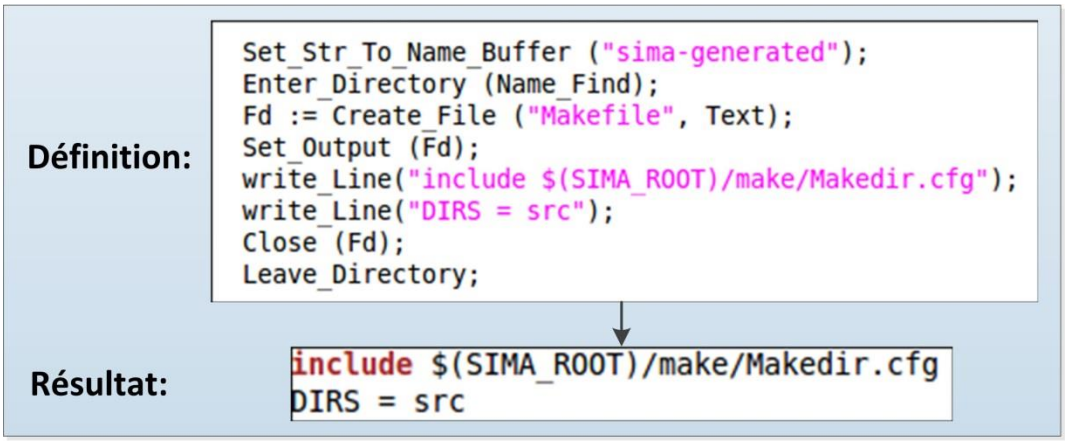

Figure 3-16: Définition d'un fichier de compilation

4. Génération de la structure de répertoires

Les modifications pour passer d'une structure de répertoires à l'autre ne sont pas compliquées quand tous les fichiers, ainsi que le code source sont prêts. La figure 3-4 montre que la structure de répertoires du système qui fonctionne sous SIMA (en bas de la figure) ressemble à celle sous POK (en haut de la figure). Il faut simplement changer les noms de répertoires et leurs positions en redéfinissant leurs «nœuds pères », en prenant la structure de répertoires d'un exemple existant de SIMA comme référence.

5. Compatibilité

Pour des raisons de compatibilité, un fichier supplémentaire qui s'appelle *pok_wrapper.c* a été créé manuellement. Ce fichier contient des fonctions qui se trouvent dans les librairies de POK,

mais qui n'existent pas dans les librairies de SIMA. Ce fichier reste encore extensible par l'usager si une autre fonction de ce type est identifiée. Plus de détails sont donnés en Annexe 3.

3.2.3 Niveau Simulink

Le flot de conception avec Simulink est reconnu par l'industrie parce qu'il offre un ensemble de solutions dans plusieurs domaines et aussi parce qu'il offre une grande flexibilité sous *MATLAB*. Comme expliqué précédemment au chapitre 2.2.2, le flot de conception avec un modèle Simulink est capable de créer des solutions pour un certain nombre de plateformes. Dans l'industrie aérospatiale, Simulink est utilisé comme un outil de conception dans de but de développer des applications en avionique. Il permet de simuler certaines fonctionnalités d'un système avionique, tels que l'environnement de vol ou des actuateurs avioniques.

Dans ce projet de maîtrise, nous visons à proposer des méthodologies pour développer efficacement des systèmes IMA. Il arrive parfois que les applications en avionique ne fonctionnent pas correctement à cause d'une mauvaise simulation de l'environnement. Cet « environnement » indique des données capturées par les capteurs et des tâches spécifiques à exécuter avec l'aide de certains actuateurs. Cet « environnement » peut être représenté et simulé par Simulink si certains composants nécessaires existent dans les librairies.

Lors de la simulation de systèmes avioniques, cet « environnement » peut être directement installé dans une ou plusieurs partitions du simulateur IMA pour compléter la simulation. Mais, cela peut causer un certain nombre de problèmes.

Tout à bord, pour que les applications d' « environnement » fonctionnent correctement sous une plateforme IMA, il faut assurer que leurs générations soient conformes au standard ARINC653 [24].

Ensuite, il faut résoudre le problème d'implémentation. Après avoir simulé le modèle, il faut voir comment le raffiner pour le porter rapidement vers un système d'exploitation partitionné tel que PikeOS. Dans de vrais systèmes avioniques complexes, les capteurs et actuateurs sont de vrais composants physiques. Ils échangent des données ou des signaux avec le module de calcul (soit IMA). Par conséquent, en phase d'implémentation, les partitions modélisant les capteurs et les actuateurs doivent être enlevées du module IMA. En fait, dans un système IMA quand le nombre de partitions change, la configuration du système et l'interconnexion entre les partitions doivent

être modifiées parce que le tableau d'ordonnancement et le nombre de ports sont changés. Et ce, davantage lorsque le système est très complexe, autrement dit, lorsqu'il y a un grand nombre de partitions et interconnexion dans un même IMA système. Le travail pour porter les applications avioniques d'une plateforme de simulation vers une plateforme d'implémentation devient donc très complexe.

Pour résoudre ces problèmes, nous proposons une approche de cosimulation.

3.3 Niveau de cosimulation

La solution proposée consiste à simuler des systèmes avioniques avec une plateforme de cosimulation entre SIMA et Simulink (Figure 3-1).

Les modèles des capteurs et des actuateurs générés par le flot de Simulink (Figure 2-6) fonctionnent indépendamment et parallèlement avec le module SIMA. Donc, ils n'ont pas à respecter obligatoirement les règles du standard ARINC653.

Dans SIMA, une partition spécifique a été créée comme un serveur /adaptateur afin de gérer la communication entre le module IMA et le modèle représentant l'environnement. La section 3.4 suivante (Section 3.4) explique pourquoi on peut résoudre le problème d'implémentation. Cependant, il faut trouver un mécanisme, nommé « bus de cosimulation », pour faire communiquer le serveur SIMA et le monde externe. Ici nous avons utilisé le protocole TCP/IP parce qu'il est de coût faible et qu'il est facile à installer sous Windows ou Linux.

Le rôle de cet adaptateur est de gérer l'échange d'informations entre SIMA et l'environnement externe. Plus précisément, c'est un adaptateur visant d'une part à redistribuer les données qui sont générées par les capteurs et transmises via le port de communication TCP/IP aux autres partitions du simulateur SIMA via les ports ARINC653, et d'autre part à collecter les signaux générés par certaines applications de SIMA et les envoyer vers les actuateurs sous format de paquets de données via le port TCP/IP. Le schéma de l'adaptateur est illustré sur la figure 3-17 qui montre le transfert de données dans une partition réservée.

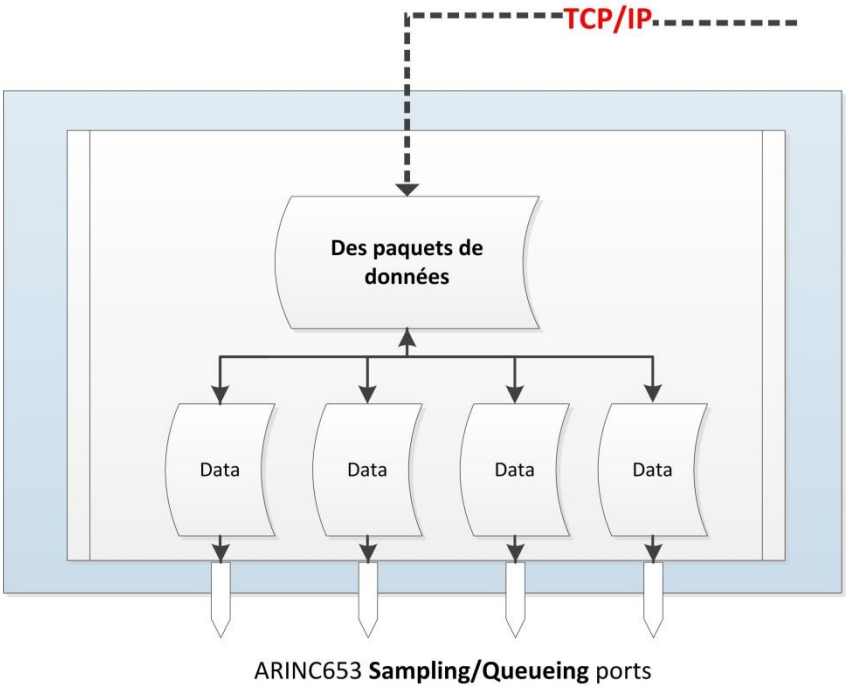

Figure 3-17: Schéma de l'adaptateur

Le développement du serveur contient principalement trois tâches. La première consiste en création d'un processus qui fonctionne sous l'environnement d'exécution ARINC653. Ce processus doit supporter l'APEX et implémenter quatre états : dormant, en attente, prêt à s'exécuter et en cours d'exécution. La deuxième tâche consiste à développer les ports ARINC653 pour l'échange d'informations entre la partition de serveur et les autres partitions. Tel que nous l'avons vu à la section 2.1.1.5, il y a deux types de ports de communication à choisir : le port de file d'attente (queueing port) et le port d'échantillonnages (sampling port). La troisième tâche consiste à développer le port de communication TCP/IP. En fait, les deux premières tâches peuvent être générées automatiquement par le générateur de code sous OCARINA (Figure 3.2). Pour la troisième tâche, il faut développer manuellement le port de communication TCP/IP.

De plus, l'exécutable développé pour Simulink et la plateforme de développement PikeOS fonctionnent sous l'OS Windows, alors que le simulateur SIMA fonctionne sous l'OS Linux. Par conséquent, le port de communication TCP/IP doit aussi supporter deux versions pour pouvoir fonctionner respectivement sous ces deux OS. Le code de version Windows est donné en Annexe 4.

3.4 Raffinement vers l'implémentation

La figure 3-18 illustre l'implémentation des systèmes vérifiés à plus haut niveau d'abstraction. Il y a autant de partitions sur la plateforme d'implémentation que sur la plateforme de simulation. Le nombre de ports reste aussi le même sur les deux plateformes. Par conséquent, la configuration des systèmes ne change pas, la relation d'interconnexion entre ces partitions ne change pas non plus. En conséquence, il est facile de porter des applications avioniques vers la plateforme d'implémentation après avoir complété l'étape de cosimulation. L'étude de cas démontrera plus en détail le processus de portage des applications.

L'environnement d'exécution ARINC653 commercial PikeOS reprend directement les applications avioniques et la configuration pour les systèmes de SIMA, il peut aussi reprendre directement les périphériques en logiciel développés par Simulink en absence de composants matériels. Finalement, il génère du « bitstream » pour une vraie plateforme IMA.

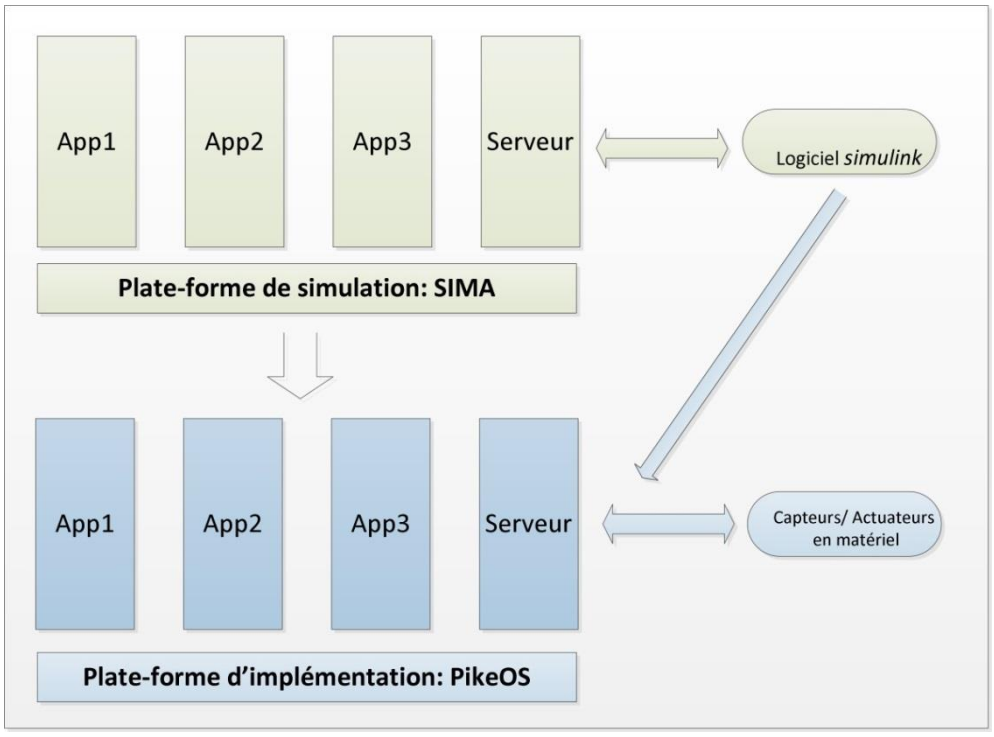

Figure 3-18: Implémentation des systèmes

Dans ce chapitre, on a proposé un ensemble de solutions qui couvre les 3 niveaux de développement de systèmes IMA : modélisation, simulation et implémentation. Les travaux nécessaires pour réaliser ce flot de conception sont aussi présentés.

CHAPITRE 4 ÉTUDE DE CAS

Dans l'étude de cas présentée dans ce chapitre, un modèle AADL et un modèle Simulink ont été développés. Le générateur OCARINA modifié pour cibler SIMA et le générateur de Simulink ont été utilisés pour valider le flot de conception. Au lieu d'applications avioniques commerciales COTS qui sont très coûteuses et difficiles à obtenir dans un contexte de recherche universitaire, une application a été développée dans notre laboratoire. La plateforme de cosimulation a été sélectionnée pour réaliser la simulation, ainsi que pour vérifier les fonctionnalités du système avionique. Pour la phase implémentation, le système vérifié a été porté vers PikeOS. Les résultats sont analysés au chapitre 5.

4.1 Niveau de modélisation, partie Simulink

4.1.1 Explication du modèle

La figure 4-1 montre la structure interne du modèle Simulink. Les ports ovales à gauche numérotés de 1 à 5 prennent les signaux d'entrée et ceux qui se trouvent à droite numérotés de 1 à 12 sont les ports de sortie. Les 6 rectangles au milieu de la figure sont des modules standards fournis par les librairies de Simulink, permettant de simuler certains paramètres importants de vol.

Les 5 signaux d'entrée sont l'altitude, la vitesse, la latitude, la longitude et l'accélérateur.

Les 12 signaux de sortie sont la température absolue, la vitesse du son, la pression statique, la densité de l'air, la pression dynamique, la vitesse du véhicule en Mach[5], la poussée, le taux de consommation de carburant, le champ magnétique, l'intensité horizontale du champ magnétique, l'intensité totale du champ magnétique et l'accélération de la pesanteur.

Le premier bloc (description de haut vers le bas) de la figure 4-1, nommé « Turbofan Engine System » [30], est un module fonctionnel qui permet de calculer en temps réel la poussée du moteur et la consommation de carburant en fonction de la position de l'accélérateur, la valeur

[5] Mach : c'est une unité pour mesurer la vitesse, il exprime le rapport de la vitesse locale d'un corps à la vitesse du son dans ce même fluide.

Mach et l'altitude du véhicule. Des paramètres initiaux tels que la poussée initiale et la poussée maximale au niveau de la mer doivent être configurés lors de la création de ce module.

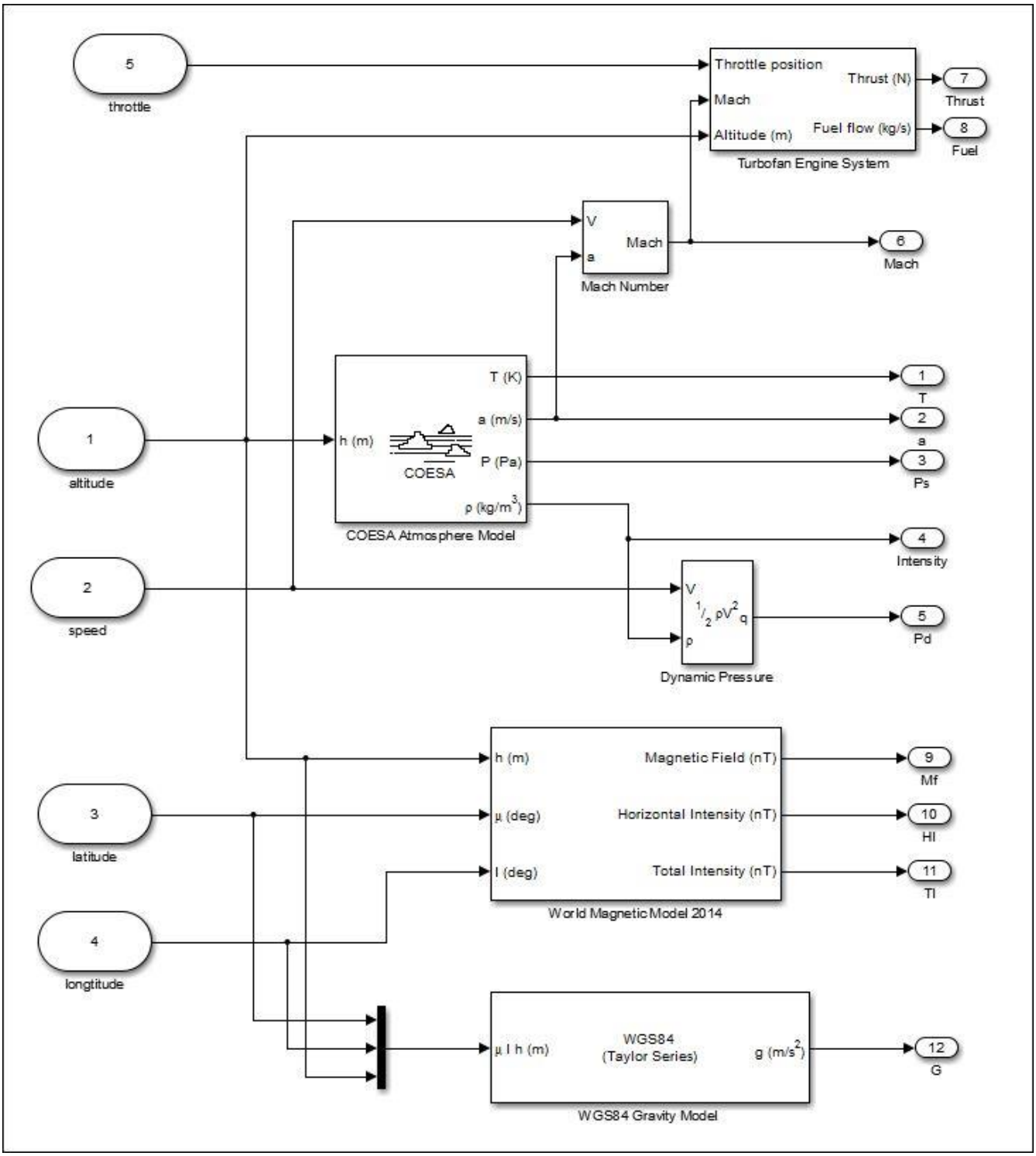

Figure 4-1: Le modèle en Simulink

Le deuxième bloc « Mach Number » [31] convertit la vitesse au numéro de Mach en prenant le rapport de la vitesse du véhicule à la vitesse du son.

Le troisième bloc « COESA Atmosphere Model » [32] implémente une représentation mathématique définie en 1976. C'est en fait un modèle atmosphérique qui numérise les valeurs absolues de la température, la vitesse du son, la pression statique et la densité de l'air en fonction de l'altitude. Ce modèle s'applique quand l'altitude est entre 0 m et 84 825 m.

Le quatrième bloc « Dynamic Pressure » [33] calcule la pression dynamique en fonction de la densité de l'air et de la vitesse du véhicule.

Le cinquième bloc « World Magnetic Model » [34] implémente une représentation mathématique pour un modèle magnétique de la terre en 2014. Le champ magnétique, l'intensité horizontale et l'intensité totale sont calculés pour un point donné (déterminé par l'altitude, la latitude et la longitude).

Le sixième bloc «WGS84 Gravity Model » [35] crée un modèle pour représenter l'accélération due à la gravité d'en point donné (déterminé par l'altitude, la latitude et la longitude).

4.1.2 L'interface usager

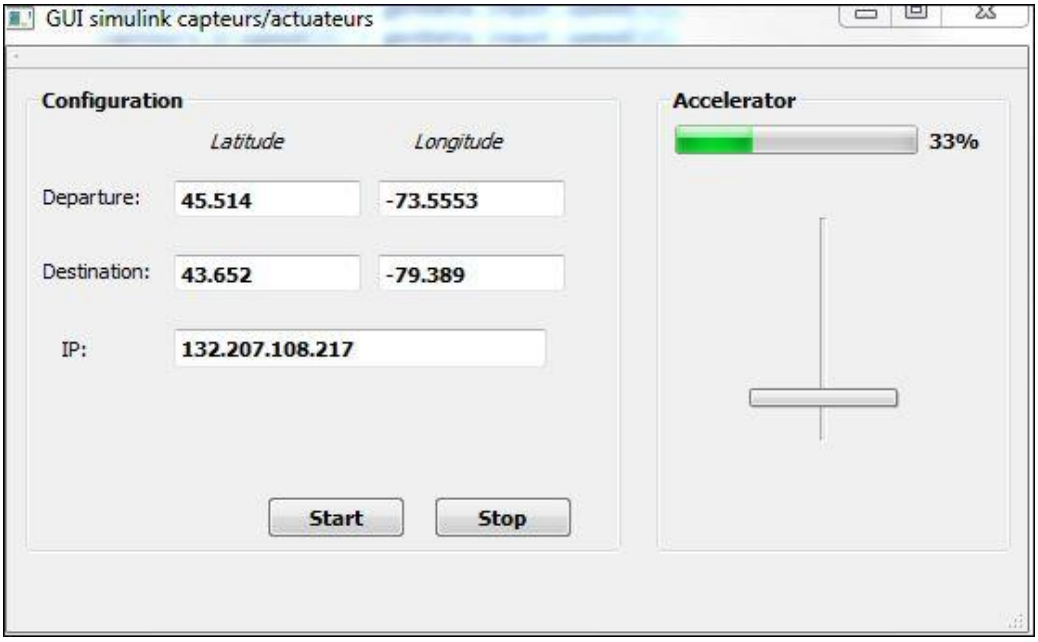

Figure 4-2: L'interface graphique d'usager

Une fois complété, le modèle Simulink de la figure 4-1 peut être compilé pour générer une solution en langage C/C++. Pour faciliter la configuration des paramètres d'entrée dans la solution générée, une interface graphique illustrée à la figure 4-2 a été développée. Les paramètres à configurer sont: la latitude et la longitude du point de départ (par défaut Montréal) et point de destination (par défaut Toronto), l'adresse IP de la machine hôte où se trouve le module IMA, et la valeur de l'accélérateur. La vitesse de l'avion est proportionnelle à la valeur de l'accélérateur (il s'agit d'un cas simplifié). Les deux boutons « Start » et « Stop » vont lancer ou arrêter la simulation de capteurs/actuateurs.

4.2 Niveau de modélisation, partie AADL

4.2.1 Présentation du modèle AADL

Le modèle AADL textuel pour cette étude de cas est présenté en Annexe 5. À la section 4.2 de la référence [8], un modèle AADL similaire est expliqué en détail. En fait, la structure de ce type de modèle AADL (ARINC653) suit toujours le même format qui peut être résumée en 6 étapes:

1. Déclaration de paquets de données

   Les données échangées entre les différentes partitions sont souvent sous format de « paquet ». Il faut donc y mentionner le type, le format et la taille des données pour définir un nouveau paquet. Dans l'exemple du modèle AADL, 4 types de paquets ont été modélisés, soit: FuelPacket, EnviPacket, FlightPacket, ServeurPacket.

2. Déclaration des partitions

   Un certain nombre de processeurs virtuels sont créés en se basant sur un seul processeur. Un processeur virtuel va représenter une partition ARINC653. Il y a cinq partitions qui ont été définies dans le modèle AADL : « serveur », « fuel », « environment », « flight » et « view ».

3. Pas d'ordonnancement du processeur

   Il faut ici spécifier aussi les caractéristiques d'ordonnancement du cadre temporel principal. Dans le modèle, la période principale a été mise à une seconde, soit 200 microsecondes pour chaque partition.

4. Contenu des partitions sous ARINC653

   C'est la plus grosse partie du modèle qui explique le contenu de chaque partition. Il faut indiquer les applications ARINC653 (« process ») qui sont liées aux partitions. Il faut aussi spécifier les tâches ou fils d'exécution qui s'exécutent à l'intérieur des applications ARINC653, ainsi que le code usager qui est lié à cette tâche.

5. La déclaration et l'implémentation de mémoires

   Il faut y définir la mémoire allouée à chaque partition, sa taille et ses droits d'accès. Par définition de l'architecture IMA, chaque partition doit avoir sa propre mémoire.

6. L'implémentation du système

   Comme dernière étape, il faut indiquer les connexions de port, l'association de partitions et applications ARINC653 et l'allocation de mémoires.

Par conséquent, un module IMA se composant de 5 partitions a été établi selon les déclarations dans le modèle AADL. Il est illustré par la figure 4-3.

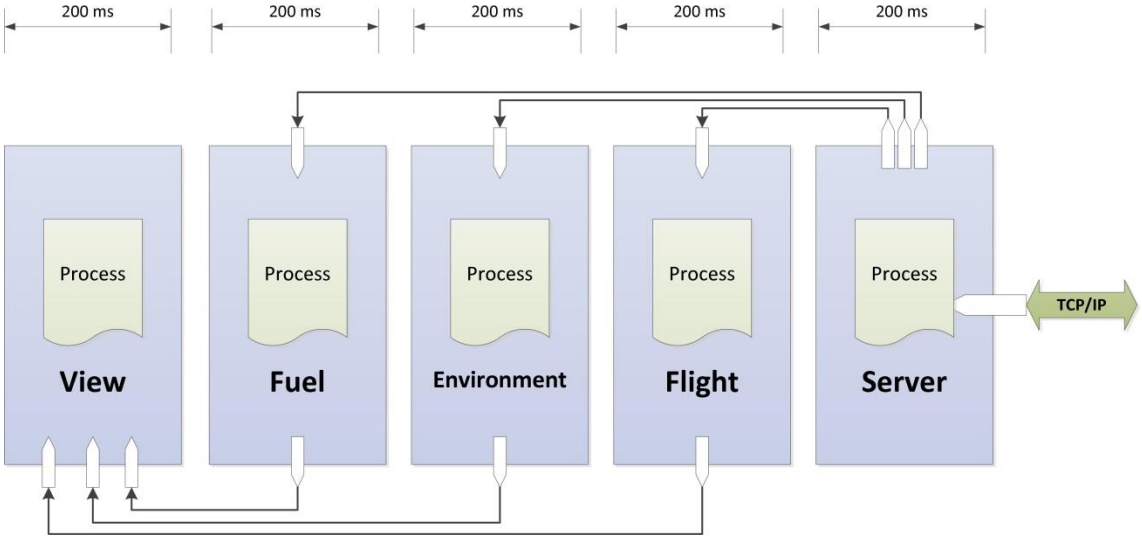

Figure 4-3: L'architecture IMA modélisée en AADL

4.2.2 Génération du code avec la nouvelle version OCARINA

Le modèle AADL expliqué ci-haut a été analysé et compilé par le générateur du code modifié - OCARINA. L'ensemble des fichiers pour l'application avionique générée se trouve dans un

répertoire appelé « sima-generated ». La structure de répertoires de cette application et tous les fichiers sont illustrés par la figure 4-4.

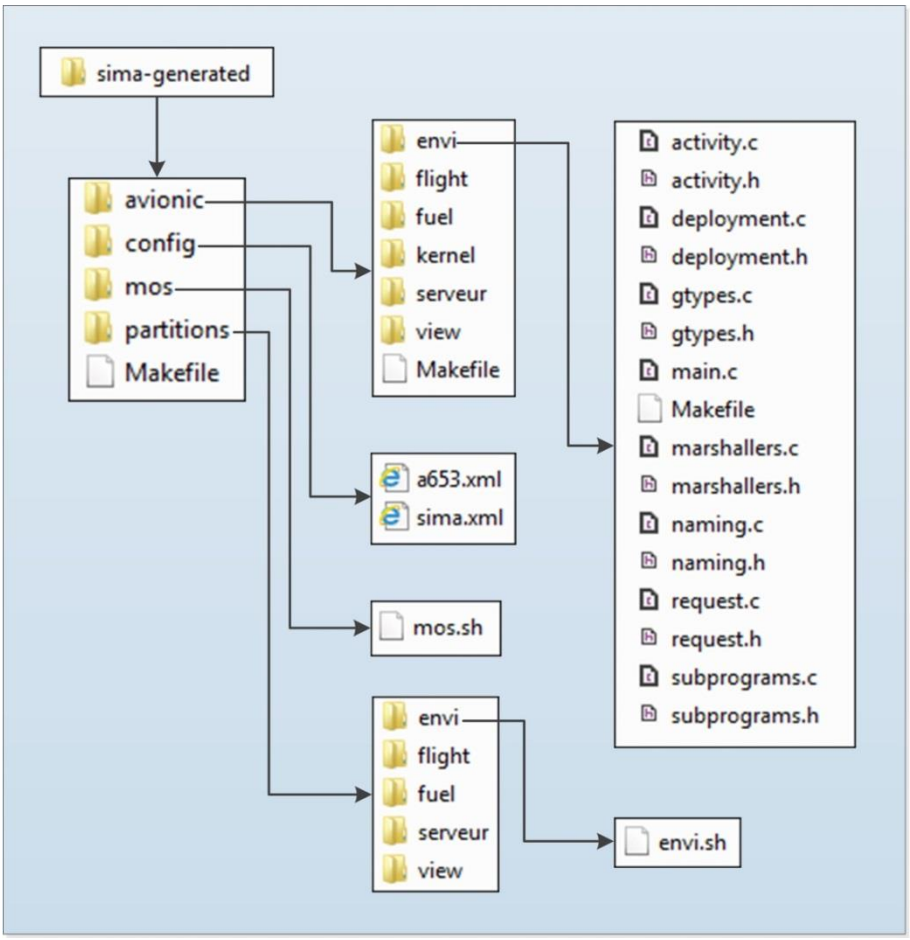

Figure 4-4: La structure de répertoires de l'application générée

4.3 Réalisation de la cosimulation

Pour compléter cette application avionique générée par OCARINA, il faut y intégrer le code d'usager où le comportement de chaque partition est défini. Concrètement, les fonctionnalités de ces partitions sont assez claires:

La partition « serveur » reçoit les paquets de données en provenance du simulateur de capteurs et les transmet ensuite à nouveau à leurs partitions destinées. Le numéro d'identificateur de paquets est affiché.

La partition « fuel » va faire la gestion du carburant d'aviation; la quantité de carburant qui reste et la distance de croisière maximale seront calculées et une alarme sera donnée si le carburant restant est insuffisant pour le voyage. De plus, la poussée et le taux de consommation de carburant seront affichés.

La partition « envi » permet de traiter les données sur l'environnement dans lequel se trouve l'avion telles que la température, la vitesse du son en temps réel, etc.

La partition « flight » calcule la vitesse de l'avion horizontale et verticale, la distance entre le point de départ et le point de destination, la distance parcourue et l'accélération due à la gravité.

La partition « view » va traiter la position en temps réel de l'avion (l'altitude, la latitude et la longitude). Cette partition est réservée afin d'ajouter un modèle GPS pour suivre la trajectoire de vol dans les travaux futures.

La plateforme matérielle de cosimulation est composée de deux ordinateurs connectés via le réseau Internet, tel qu'illustré par la figure 4-5. Le logiciel SIMA a été installé sur un PC et le logiciel de capteurs/actuateurs va s'exécuter sur un autre. Les solutions créées par les deux flots de conception peuvent être testées sur ces deux ordinateurs. Les résultats de cosimulation seront présentés au chapitre 5.

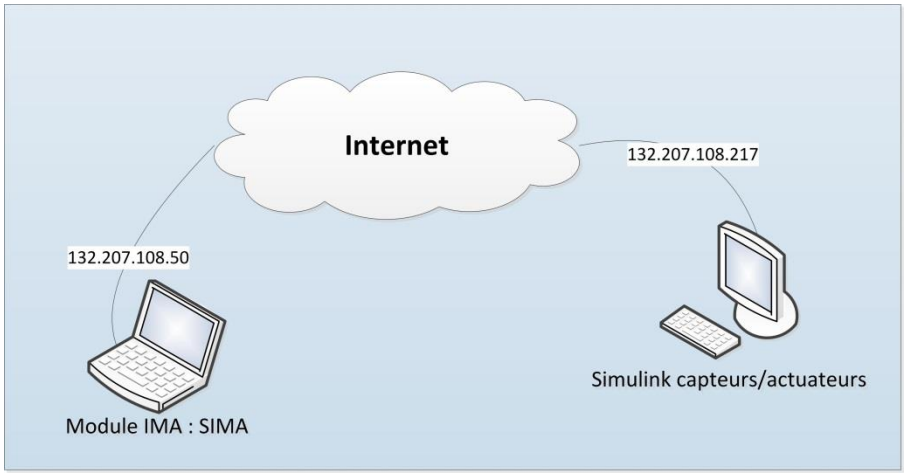

Figure 4-5: La plateforme matérielle de cosimulation

4.4 Implémentation du système avionique

Comme environnement de développement peu coûteux, la plateforme de cosimulation décrite ci-haut peut être utilisée comme une solution intéressante afin de : 1) développer et tester les fonctionnalités d'usager pour les applications avioniques, 2) trouver une bonne configuration du système et 3) simuler l'ensemble du système. Ces efforts ne sont pas vains, par contre, ils sont très utiles pour le développement des systèmes dans la phase de l'implémentation grâce à la réutilisation des fichiers.

Une grande partie des fichiers dans l'application ARINC653 de SIMA peuvent être repris pour PikeOS. Ceux-ci comprennent le code d'usager vérifié, ainsi que le code de démarrage généré permettant de créer des processus, des ports de communication et l'intégration d'autres fonctionnalités provenant de l'API ARINC653. De plus, même si les fichiers pour la configuration de système (*a653.XML et sima.XML*) ne peuvent pas être directement utilisés, cependant, ces fichiers peuvent être de bonnes références pour l'ordonnancement du système, pour la connexion entre partitions, etc.

Tel que mentionné dans la section 2.1.3.1, PikeOS a son propre environnement de développement: CODEO. Ce dernier permet de faire la gestion du projet où se trouvent le code d'usager et le code de démarrage, de configurer le système via un éditeur graphique. Cet assembleur graphique permet d'ajouter et de supprimer rapidement des partitions, des applications et des services comme les pilotes en référençant les fichiers de configuration générés pour SIMA.

4.4.1 Raffinement du code

Sur la plateforme CODEO de PikeOS, des projets APEX peuvent être créés rapidement. La figure 4-6 donne un exemple du projet PikeOS (*serveur.app*). Le raffinement du code consiste à copier le code de démarrage et le code d'usager d'une partition SIMA dans un projet PikeOS. Comme la structure de librairies est organisée différemment, le fichier en-tête *#include <a653.h>* doit être remplacé selon les besoins par *#include <apex_process.h>, #include <apex_partition.h>, #include <apex_queuingports.h>, etc.* En plus, PikeOS a son propre gestionnaire d'erreurs. Il faut donc le définir et le rajouter dans le code. SIMA et PikeOS

respectent tout le standard ARINC653, ainsi, ils appellent les mêmes fonctions de l'API ARINC653 pour définir un processus, créer un port, envoyer ou recevoir des paquets de données. Par conséquent, le reste du code qui définit des fonctionnalités n'a pas besoin d'être modifié. On doit donc simplement répéter ces étapes pour les 5 partitions de SIMA dans le but de créer 5 projets APEX de PikeOS. Ces derniers sont des partitions qui s'exécuteront alors sur la plateforme PikeOS.

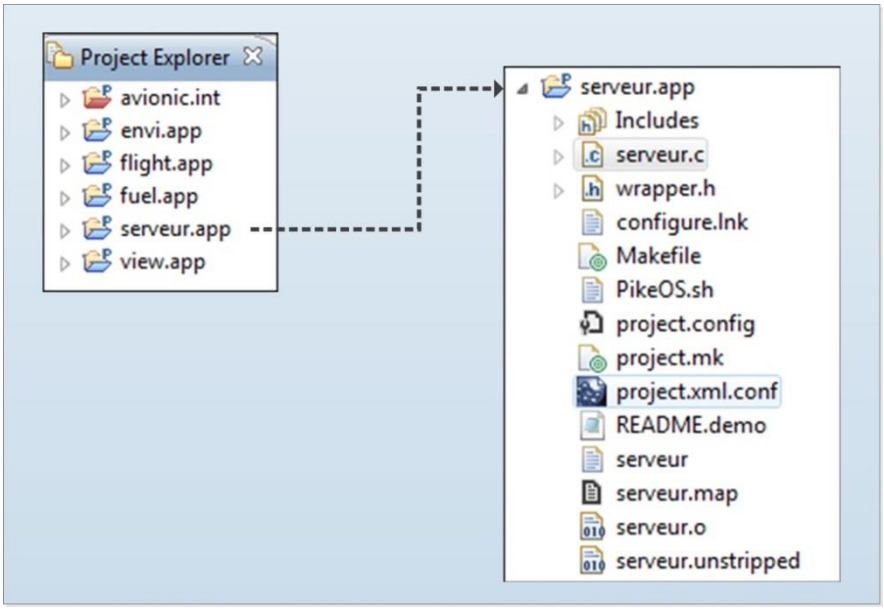

Figure 4-6: Les projets sous CODEO

4.4.2 Raffinement des configurations

Comme précisé précédemment, les fichiers de configuration sous SIMA, soit *a653.XML* et *sima.XML*, peuvent être référés afin de récupérer des paramètres de configuration.

D'abord, pour définir les ports de communication, il est possible de récupérer des paramètres utiles tels que le nom de port, la taille de message, la direction de port (destination/source) et le type de port (ports de file d'attente ou d'échantillonnages). Une fois que tous les ports sont bien configurés, l'éditeur graphique de CODEO permet de créer des canaux de communication en reliant les ports par l'intermédiaire de la souris, tel qu'illustré à la figure 4-7.

Pour configurer les mémoires de partition, nous reprenons les propriétés telles que la taille et la lecture/écriture. Ensuite, en utilisant encore la souris, nous les connectons aux mémoires partagées (Figure 4-8).

CODEO attribue un numéro d'identification à chaque partition. Dans cette étude de cas, nous obtenons 1 pour la partition « environnement », 2 pour « flight », 3 pour « fuel », 4 pour « serveur » et 5 pour « view ». En respectant les configurations pour SIMA, une période de 200 millisecondes est allouée à chaque partition, et l'ordre de l'ordonnancement en fonction des numéros de tâches devient alors 4-3-1-2-5 (Figure 4-9).

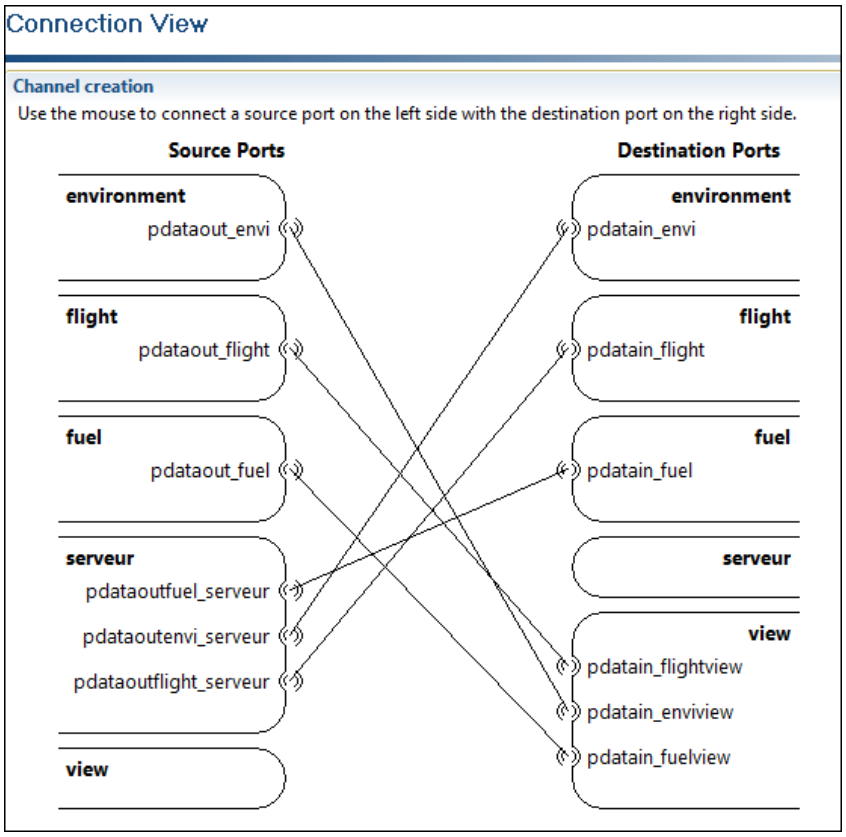

Figure 4-7: Les canaux de communications

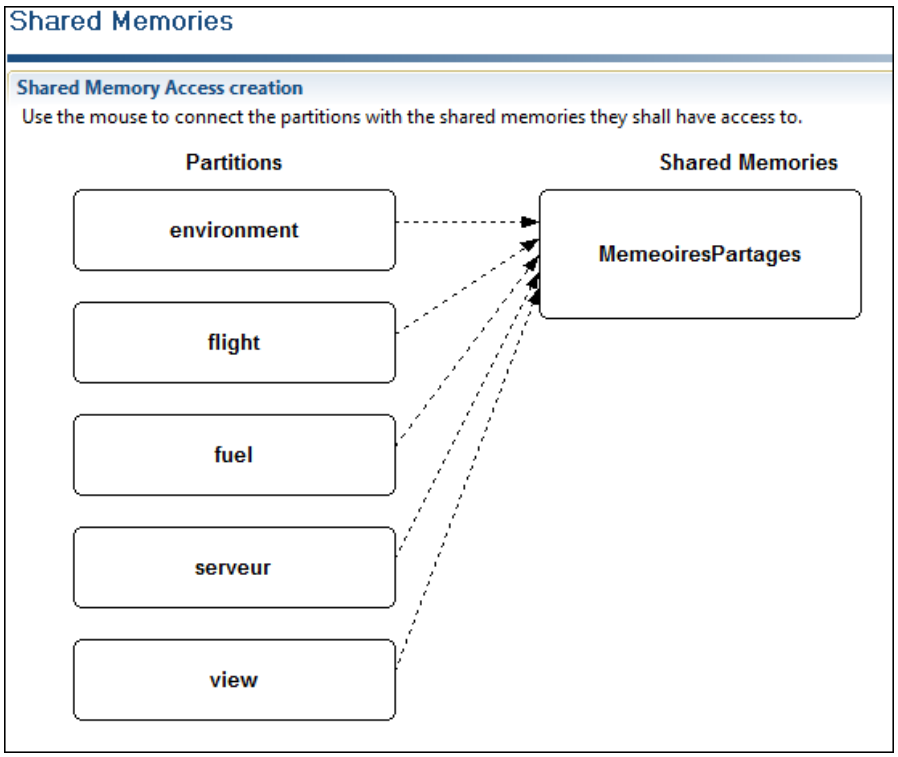

Figure 4-8: La configuration des mémoires

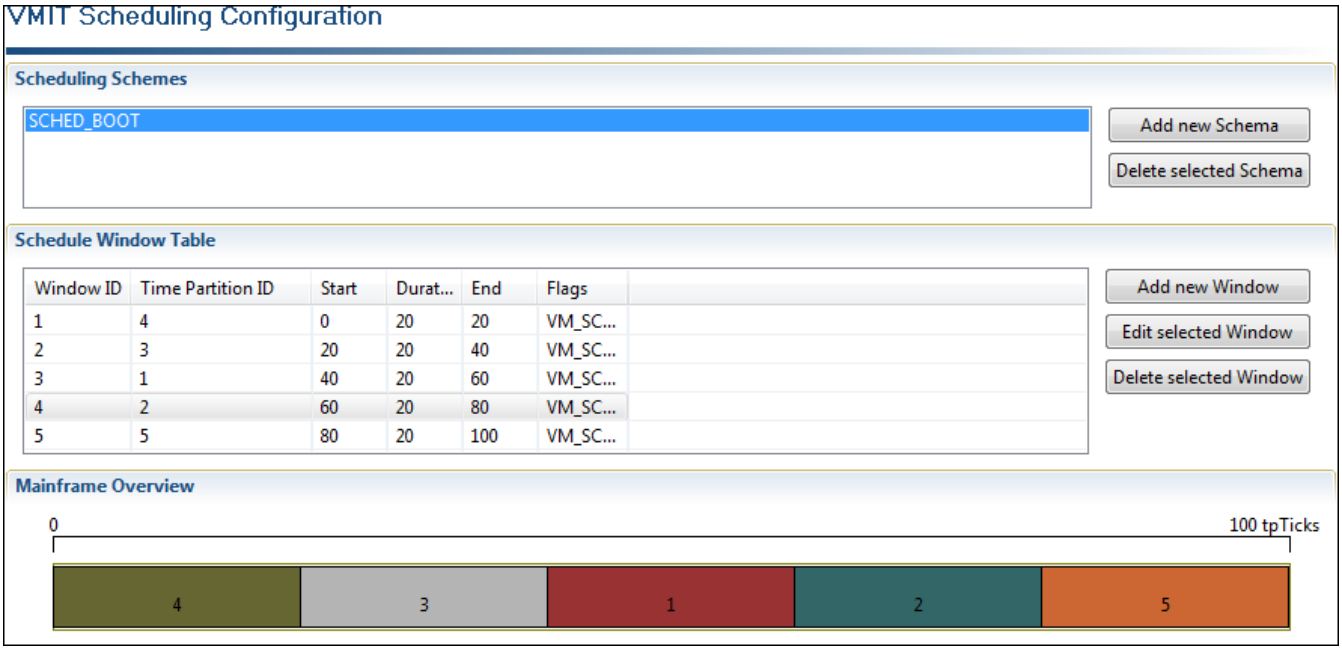

Figure 4-9: L'ordonnancement des partitions

4.4.3 Implémentation finale

Comme le code d'usager et le code de démarrage sont déjà vérifiés sur la plateforme de cosimulation, qu'ils peuvent être repris par PikeOS avec peu de modifications, et que les fichiers de configuration de SIMA sont repris comme références, le temps requis pour développer un système avionique est grandement diminué. Cependant, dans un contexte industriel, la certification doit encore être faite pour s'assurer de la qualité du système avant son implémentation. La plateforme CODEO va générer la solution ROM pour une plateforme matérielle IMA pouvant être téléchargée sur une carte ciblée.

Le bus de cosimulation TCP/IP peut alors être remplacé par le bus avionique AFDX, les modèles de capteurs et d'actuateurs développés sous Simulink peuvent alors être remplacés par leurs équivalents physiques. Néanmoins, pour tester le fonctionnement de la carte ciblée, le bus de cosimulation TCP/IP et les périphériques en logiciel peuvent encore être réutilisés.

CHAPITRE 5 EXPÉRIMENTATION

Ce chapitre présente les résultats obtenus à partir de l'étude de cas présentée au chapitre 4. Les résultats observés de cosimulation sont analysés. Les avantages et désavantages de cet ensemble de solutions apportées sont discutés.

5.1 Résultats observés

5.1.1 Cosimulation

Après que les paramètres soient correctement configurés et que les deux systèmes de la cosimulation soient lancés, quelques captures d'écran ont été prises pendant l'exécution de la plateforme de cosimulation (figures 5-1, 5-2, 5-3) afin de démontrer le bon fonctionnement de cette dernière.

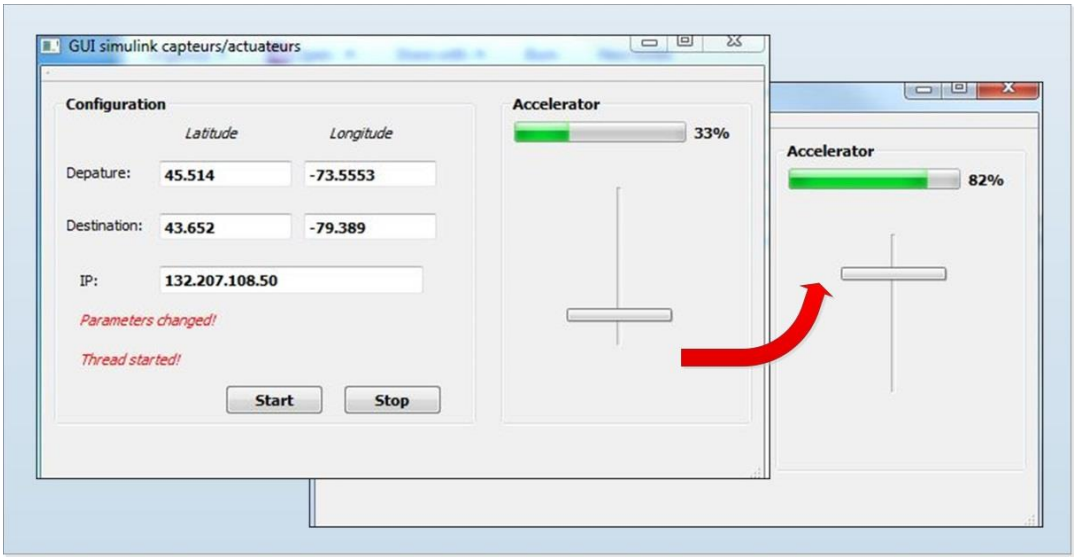

Figure 5-1: Valeur d'accélérateur de 33% à 82%

Dans cet exemple, un vol en provenance de Montréal et à destination de Toronto a été simulé. La figure 5-1 indique que la valeur de l'accélérateur est passée de 33% à 82%. À la suite de l'accélération, la capture d'écran sur la figure 5-3 montre la mise à jour de paramètres par rapport à la capture d'écran de la figure 5-2: la poussée des moteurs est passée de 7817 newtons à 21281 newtons. La vitesse horizontale a été augmentée de 258 km/h à 643 km/h. Lorsque l'avion monte, les valeurs des paramètres tels que la température, la pression statique, la vitesse du son, et l'accélération due à la gravité diminuent. Pendant que la distance parcourue s'accumule, la

quantité de carburant restant diminue, par contre, la position de l'avion qui est représentée par la latitude et la longitude approche de plus en plus la destination. En bas des figures 5-2 et 5-3, la zone « test » montre l'état de l'ordonnancement des 5 partitions. Chaque partition prend 0.2 seconde et il est en mode d'opération « NORMAL ». Bref, il semble bien que la plateforme de cosimulation fonctionne correctement pour cette première application.

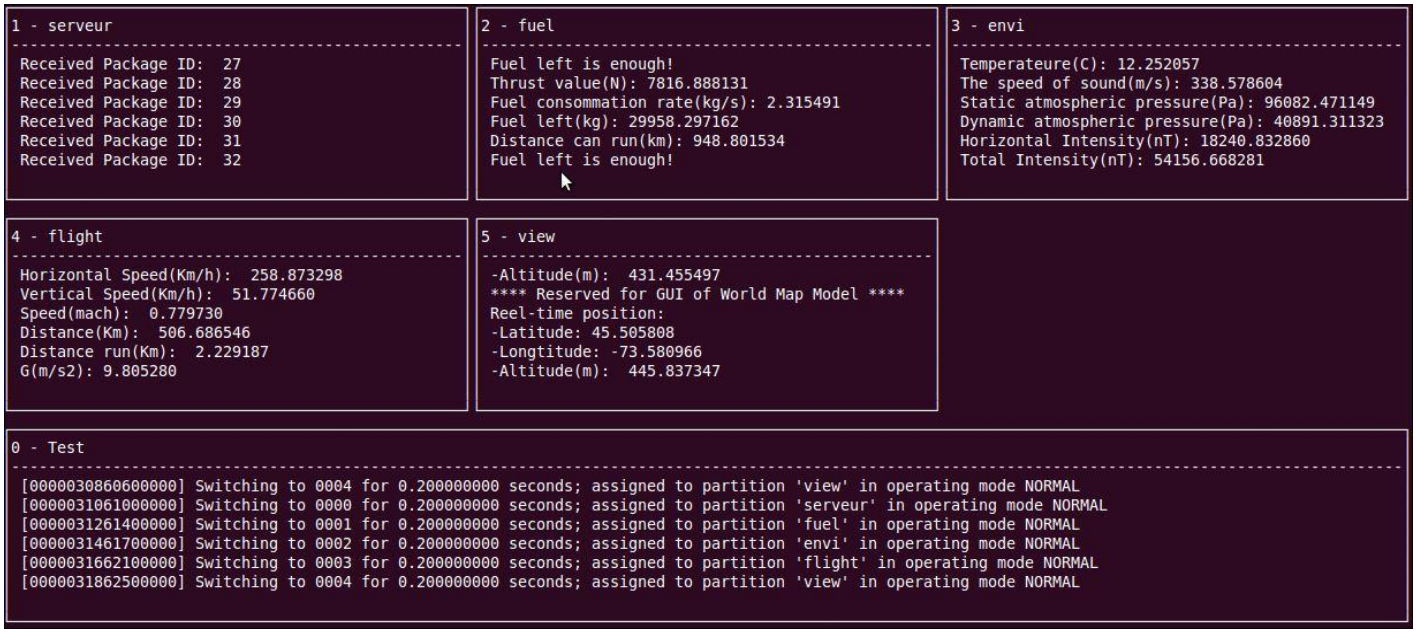

Figure 5-2: Capture d'écran de SIMA (Accélérateur à 33%)

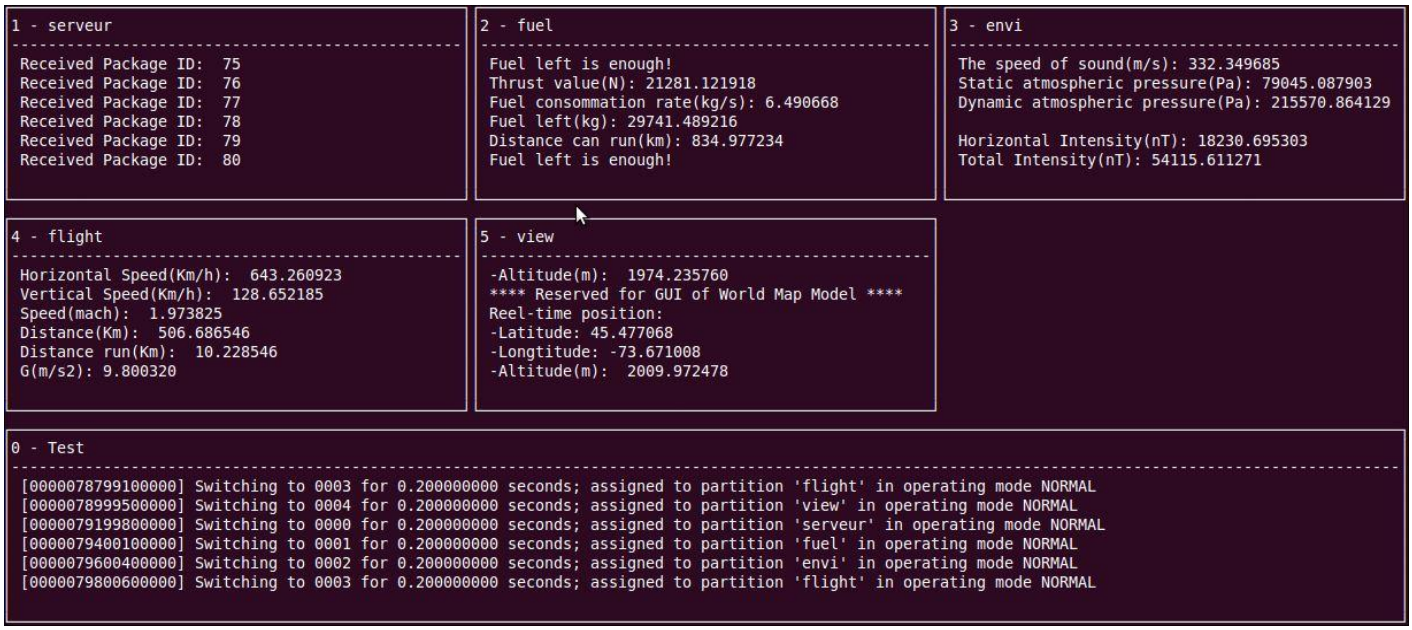

Figure 5-3: Capture d'écran de SIMA (Accélérateur à 82%)

5.1.2 Raffinement du modèle vers PikeOS

Une fois que cette même application a été portée sur la plateforme PikeOS, CODEO peut alors assembler tous les composants nécessaires (des librairies, des applications, le noyau, des propriétés, etc.) pour générer une image ROM. En l'absence de plateforme matérielle, la solution peut s'exécuter sur l'émulateur de machine QEMU (Section 1.3-iii). Puisque la licence PikeOS utilisée dans ce projet est une licence de type académique, le support au niveau contenu des librairies est assez limité. À titre d'exemple, la librairie pour l'en-tête *winsock2.h* n'est pas supportée, par conséquent, la communication entre le système IMA développé par PikeOS et les capteurs sous Simulink ne peut pas être testée comme prévu. Autrement dit, bien qu'une cosimulation SIMA/Simulink ait été réalisée, à cause d'un problème d'accès à la librairie, il nous a été impossible d'effectuer la cosimulation QEMU/Simulink.

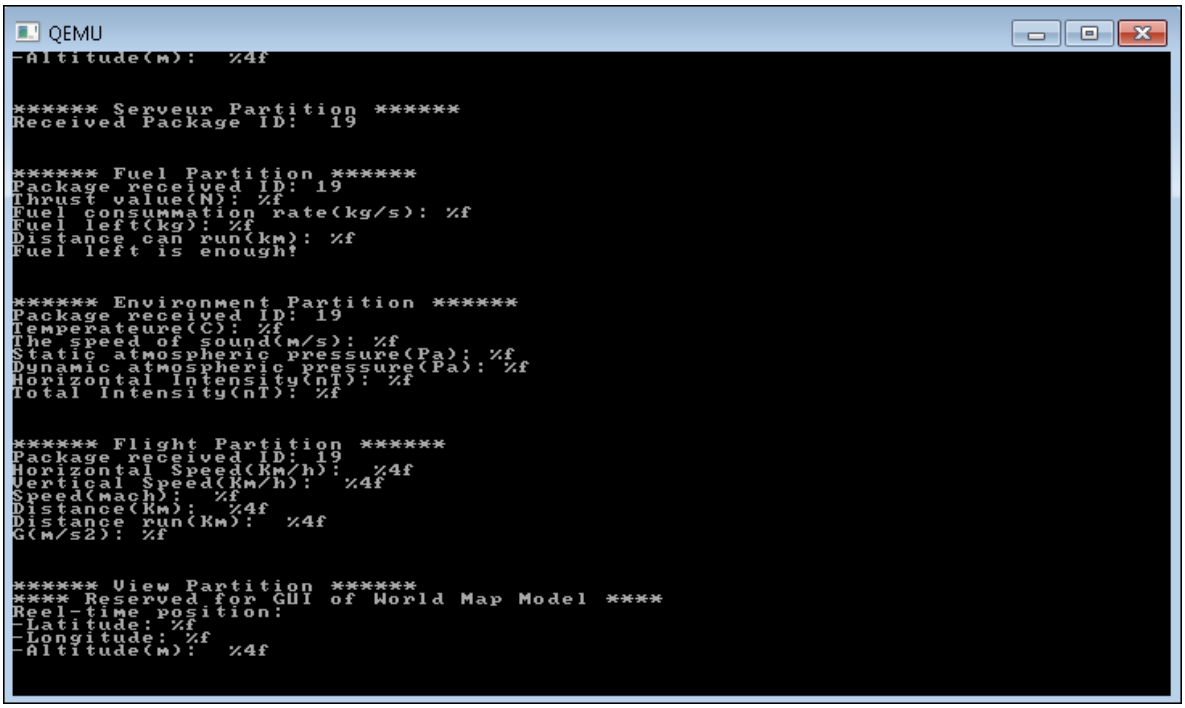

Figure 5-4: Capture d'écran de PikeOS

Cependant, les résultats donnés par QEMU, illustrés sur la figure 5-4, démontrent que l'application fonctionne correctement sous PikeOS: les 5 partitions peuvent être démarrées avec réussite et elles s'exécutent dans le même ordre et avec les mêmes périodes que sur SIMA en affichant les mêmes messages ; les autres partitions reçoivent un ID de paquet envoyé par la partition « serveur », ce qui indique que les canaux de communication fonctionnent bien. De plus,

comme les fonctionnalités du code sont déjà vérifiées sur SIMA, on peut faire l'hypothèse que les travaux de raffinement fonctionnent bien pour cette première application.

5.2 Analyse des résultats obtenus

Dans l'étude de cas présentée précédemment, deux étapes du flot de conception ont été validées avec une première application. Cet ensemble de solutions couvre le niveau de modélisation, le niveau de cosimulation et le niveau de l'implémentation du processus de développement des systèmes IMA. Il couvre également les étapes pour passer d'un niveau à l'autre, soit : 1) le niveau de génération de code sous OCARINA et de génération d'un exécutable Simulink pour la cosimulation et 2) le raffinement pour passer d'un modèle à une implémentation PikeOS simulable sous QEMU et éventuellement exécutable sur une carte physique. Bien qu'il y ait encore des points à améliorer au niveau de l'automatisation du flot, les résultats expérimentaux présentés à la section 5.2.1 nous indiquent un gain d'au moins 50% au niveau du temps de conception.

5.2.1 Avantages des solutions proposées

1. Efficacité en termes de temps

Tout d'abord, les deux flots de conceptions sont basés sur la méthodologie de l'ingénierie dirigée par les modèles qui, comme mentionnée plutôt, est considérée comme une solution très efficace pour développer les systèmes avioniques critiques. Le développement de la partie textuelle du modèle AADL est rapide, car la description d'une architecture IMA en AADL suit toujours le même schéma, ce qui favorise la réutilisation. L'effort est surtout dans la première description de modèle AADL. Autrement dit, pour concevoir un nouveau modèle IMA à partir d'un modèle existant sous AADL, l'effort se concentre surtout sur l'ajout ou le retrait des partitions, des processus, des programmes d'usager, des ports et des connexions. Comme indiqué dans le tableau 5.1, une fois que le modèle AADL est conçu, le processus de génération du code via OCARINA prend quelques secondes. De même, en ce qui concerne la création du modèle Simulink (modélisation et génération de code), grâce aux nombreux composants graphiques disponibles en librairie, le temps de conception est également très rapide. Pour compléter cette expérimentation, il aurait fallu demander à un concepteur indépendant de réaliser cette étude de cas avec une approche traditionnelle (e.g., manuellement) pour mieux apprécier les gains de

temps en conception. Malheureusement, par manque de ressources (eg., personnel qualifié, applications avioniques réelles), ceci n'a pas pu être complété. Mais toutefois, nous estimons cet effort à plusieurs heures (possiblement en jours pour le débogage). Finalement, la même comparaison aurait pu être faite en ce qui concerne la qualité du code des fichiers de configuration, mais puisque notre approche supporte le standard ARINC653, et par définition de ce même standard, la qualité générée devrait s'approcher de celle d'une conception manuelle.

Tableau 5.1: Différents temps de développement obtenu avec notre flot de conception

Tâche à réaliser	
Développement d'une application avionique pour SIMA	• Modèle AADL: 2 heures • Génération du code sous OCARINA: 5 secondes • Intégration du code usager : 1 heure
Développement du simulateur pour les capteurs/actuateurs	• Modèle Simulink : 1 heure • Génération du code : 5 secondes
Raffinement pour une implémentation sur PikeOS	• Portage du code source et de la configuration : 4 heures.

En ce qui concerne la partie raffinement afin de réaliser l'implémentation sous PikeOS, notons qu'une grande partie de l'application sous SIMA peut être directement réutilisée ou réutilisée en grande partie. Plus particulièrement, on pense à la validation fonctionnelle du code et à la configuration de système d'exploitation. Il faut aussi noter qu'une simulation sous SIMA (simulation purement fonctionnelle) est en général plus rapide qu'une simulation sous QEMU (simulation au niveau du jeu d'instructions du processeur). Il est donc plus facile d'obtenir une couverture fonctionnelle représentative sous SIMA, de sorte que seulement un sous-ensemble de cette couverture peut être réappliqué sous QEMU (afin d'assurer une certaine équivalence des deux modèles). Il est important de se souvenir ici que dans un système embarqué, une partie importante des bogues, voire la majorité, sont généralement de nature fonctionnelle [43] et peuvent donc être détectés par un simulateur tel que SIMA. Dans notre étude de cas (3^{ième} ligne du tableau 5.1), ce travail a demandé 4 heures.

Enfin, le module SIMA et le simulateur de capteurs/actuateurs sont indépendants l'un de l'autre, il est donc possible de développer parallèlement les deux parties de la plateforme de cosimulation conjointement.

2. Efficacité en termes de coût

SIMA a été choisi comme un simulateur dans le but de remplacer les environnements de développement commerciaux qui sont très coûteux (de l'ordre de la centaine de milliers de dollars, les valeurs exactes ne sont pas accessibles au public). En fait, SIMA est considéré comme une alternative efficace en termes de coût (quelques milliers de dollars par an) pour déboguer, tester et simuler les systèmes IMA tout en offrant quand même un bon degré de précision par rapport à la plateforme matérielle. Le développement des systèmes avioniques complexes doit être réalisé par de grande équipe. L'utilisation de licences SIMA en première ligne devrait faire diminuer la quantité de licences requises de type OS 653 (Sysgo, WindRiver, Green Hills, etc.). Ce gain (e.g., d'un facteur d'au moins 10) sera grandement apprécié dans la petite et moyenne entreprise.

Finalement, le coût d'une licence Simulink est relativement abordable (quelques milliers de dollars par an pour une version standard, environ cent dollars pour une version d'étudiant).

3. Cosimulation

La plateforme de cosimulation peut profiter des avantages de chaque unité de simulation, tel qu'indiqué plutôt dans le chapitre 1.1.3, la capacité de simulation est donc augmentée. La plateforme de cosimulation proposée ci-haut permet de simuler à la fois le module IMA (unité de traitement) et ses périphériques grâce à la conformité à la norme ARINC653 de SIMA et la facilité de modélisation des périphériques sous Simulink.

De plus, puisque le bus de cosimulation respecte le protocole TCP/IP, le travail de conception peut se faire conjointement et à distance (figure 4-5).

4. Portabilité

Comme une partition supplémentaire est réservée dans de but de gérer la communication entre le module IMA et le monde externe, l'application générée pour SIMA a pu être rapidement portée vers la plateforme PikeOS. Il ne reste que l'adaptateur bien localisé dans cette partition supplémentaire à modifier.

5.2.2 Désavantages des solutions proposées

1. Contraintes de temps faible

Le bus de cosimulation, basé sur le protocole TCP/IP, n'est pas temps réel. Il peut donc se produire un retard de transfert de données. Il risque aussi d'y avoir un problème de réseau ou de perte de données surtout quand la cosimulation est faite à distance. Cependant, SIMA est un simulateur non critique (non safety-critical), car il s'exécute sur un système d'exploitation Linux comportant un noyau en temps réel. Au niveau de la simulation, la plateforme de cosimulation peut quand même simuler les caractéristiques non temps réel pour les fonctionnalités d'usager. Ce qui représente somme toute une partie importante de la spécification. À l'avenir, il serait souhaitable d'avoir accès à un modèle d'ARINC664.

2. Limite des librairies (Simulink)

Simulink est souvent utilisé pour faire la modélisation dans plusieurs domaines tels que l'aérospatiale, l'informatique et l'automobile. Il possède une grande librairie. Mais, le nombre de composants prêts à utiliser dans la librairie d' « Aerospace blocket » est quand même limité. Autrement dit, il serait possible de ne pas trouver certains composants standards dans la librairie lors de la conception des systèmes avioniques complexes.

3. Limite des annexes (AADL)

Dans les modèles AADL, seulement les caractéristiques définies dans les annexes du langage peuvent être utilisées pour décrire une architecture ARINC653. Ce sont des composants standards. Par contre, pour ceux qui ne se trouvent pas dans ces annexes, le concepteur doit définir par lui-même un nouveau composant ou caractéristique. Par exemple, dans la définition de processus de SIMA, un attribut « time_capacity » n'est pas décrit dans l'annexe [37]. Par conséquent, cet attribut n'est pas modélisable en AADL. Au lieu d'étendre ses annexes, une autre solution consiste à rajouter manuellement cet attribut non modélisable dans le code généré.

CONCLUSION

Dans le cadre du projet AVIO509, un ensemble d'outils a été proposé pour aider au développement des systèmes avioniques modulaires intégrés dans ce mémoire, y compris un flot de conception et une plateforme de cosimulation. Une étude de cas a été réalisée pour faire une première validation de ce flot.

Ce flot de conception suit le concept de l'ingénierie dirigée par les modèles. La première partie du flot consiste à représenter à la fois le matériel et le logiciel de l'architecture IMA par les composants du langage de modélisation AADL. Ensuite, des travaux de modifications ont été faits sur le générateur du code OCARINA afin de pouvoir générer automatiquement des applications à partir du modèle AADL pour un simulateur commercial de système ARINC653 appelé SIMA. Ce simulateur est une alternative peu coûteuse aux environnements de développement commerciaux très dispendieux. La deuxième partie du flot consiste à modéliser l'environnement externe, soit les capteurs et actuateurs, en utilisant l'outil Simulink qui fournit des composants logiciels prêts à utiliser. Le générateur de code intégré dans Simulink permet de générer du code C ou C++ à partir du modèle Simulink. La solution générée peut fonctionner sur différents OS tels que Linux ou Windows dans le but de simuler les fonctionnalités des capteurs et actuateurs avioniques. Il est prouvé que ces deux parties de flot sont efficaces en termes de temps en utilisant la génération automatique qui accélère le développement des systèmes et élimine les erreurs humaines.

Une plateforme de cosimulation composée du simulateur SIMA et du simulateur de l'environnement externe communiquant par le protocole TCP/IP permet de simuler des systèmes avioniques plus complexes. Pour que l'application vérifiée avec SIMA puisse être portée rapidement vers un environnement de développement commercial tel que PikeOS, il est proposé de réserver une partition qui s'occupe exclusivement de la communication entre le module IMA et le monde externe. De ce fait, la configuration de systèmes et surtout les connexions entre les différentes partitions restent inchangées pendant la migration de l'application. Les capteurs ou actuateurs standards développés par Simulink peuvent être remplacés directement par les vrais matériaux correspondants, et le bus TCP/IP peut être remplacé par le bus AFDX. Par conséquent, le processus de l'implémentation est accéléré.

Une étude de cas a été faite par le développement d'une application avionique simple en utilisant le flot de conception mentionné ci-haut. Cette application simule correctement sur la plateforme de cosimulation. Des avantages et désavantages ont été soulignés suite à cette étude de cas. Cependant, la cosimulation de cette première application a démontré une première validation du flot de conception et son efficacité. Les résultats obtenus démontrent aussi l'atteinte de nos objectifs.

En ce qui concerne les travaux futurs, une très grande possibilité existe. D'abord, il faudrait valider l'approche avec un plus grand nombre d'études de cas. Ensuite, d'autres travaux pourraient consister à élargir la librairie avionique de Simulink dans le but de fournir plus de composants standards. De même on pourrait étendre l'annexe du langage AADL pour pouvoir décrire plus de caractéristiques de systèmes IMA. Il serait aussi pertinent de continuer à apporter des modifications au générateur OCARINA pour réaliser d'autres fonctions telles que la création d'une interface de communication avec le monde extérieur et la prise de décision automatique pour le partitionnement. Aussi, des tests plus rigoureux sur une plateforme matérielle avec des applications COTS restent à faire pour valider ces méthodologies.

Bref, tous les efforts mis par l'équipe de recherche et toutes les contributions de ce travail ont eu pour but de réduire les coûts de développement, du poids, de la puissance dissipée et des appareils pour l'industrie aérospatiale. Cet ensemble de solutions efficaces en termes de temps et de coût pourrait être pris comme référence par les petites entreprises qui ont un budget très limité et qui développent des applications avioniques de type COTS. Cela peut être aussi utile pour les grosses compagnies qui cherchent l'efficacité pour l'implémentation de systèmes IMA. Enfin, une extension possible de ces solutions pourrait aussi être considérée pour les domaines tels que la mécanique et l'automobile.

BIBLIOGRAPHIE

[1]. P. Bieber, F. Boniol, M. Boyer, E. Noulard et C. Pagetti, “New challenges for future avionics architectures,” *AerospaceLab*, vol. 04-11, 2012.

[2]. J. Windor et K. Hjortnaes, “Time and space partitioning in spacecraft avionics,” Dans *3rd IEEE International Conference on Space Mission Challenges for Information Technology*, 2012, pp.13-20.

[3]. École de technologie supérieure, “AREXIMAX – Architectural Exploration in Integrated Modular Avionics Systems,” [En ligne], disponible: http://areximas.etsmtl.ca (Consulté le 15 Juin 2014).

[4]. Aeronautical Radio Inc., “Avionics Application Software Standard Interface PART 1-Required Services,” Airlines Electronic Engineering Committee, ARINC653 P1-3, 2010.

[5]. Aeronautical Radio Inc., “Avionics Application Software Standard Interface PART 2-Extended Services, ” Airlines Electronic Engineering Committee, ARINC653P2-1, 2008.

[6]. Aeronautical Radio Inc., “Avionics Application Software Standard Interface PART 3-Conformity Test Specification,” Airlines Electronic Engineering Committee, ARINC653P3, 2006.

[7]. Aeronautical Radio Inc., “Part 1: Systems Concepts and Overview”, Airlines Electronic Engineering Committee, ARINC664, 2002.

[8]. J. SAVARD, “Intégration d’un simulateur de partitionnement spatial et temporal à un flot de conception base sur les modèles,” *École polytechnique de Montréal*, Montréal, Qc, Canada, 2012

[9]. M. A. Abdul Rahman, M. Mizukawa, “Modeling and design of mechatronics system with SysML, Simscape and Simulink,” Dans *IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, 2013, pp.1767-73.

[10]. CMC Electronique, “Portrait de la Société” [En ligne], disponible: http://www.esterline.com/avionicssystems/fr-ca/aboutcmc/companyprofile.aspx (Consulté le 20 Juin 2014).

[11]. CAE Électronique, “A propre de CAE,” [En ligne], disponible: http://www.cae.com/a-propos-de-cae/ (Consulté le 20 Juin 2014).

[12]. S. J. Mellor, A. N. Clark et T. Futagami, “Model-driven development,” *IEEE Software*, vol. 20, 5, pp.14-18, 2003.

[13]. S. Santos, J. Rufino, T. Schoofs et J. Windsor, “A portable ARINC653 standard interface,” dans *Digital Avionics Systems Conference*, 2008, pp. 1.E.2-1 - 1.E.2-7.

[14]. GMV IMA Groupe de recherche. “SIMA command line tools-application development and configuration guide,” GMV, version 0.9.2.5, 2010.

[15]. SYSGO AG. “PikeOS Fundamentals”, PikeOS v3.2, version S2872-1.7, 2005-2011

[16]. SYSGO Embedded Innovations, “PikeOS Hypervisor RTOS Technology,” [En ligne], disponible: http://www.sysgo.com/ (Consulté le 2 juillet 2014).

[17]. J. Delange, A. Plantec, L. Pautet, M. Kerboeuf, F. Singhoff et F. Kordon, “Validate, simulate, and implement ARINC653 systems using the AADL,” *Ada Letters*, vol. 29, 3, pp.31-44, 2009.

[18].D. J. Suo, J. X. An, J. H. Zhu, “AADL-based modeling and TPN-based verificaiton of reconfiguration in integrated modular avionics,” dans *Software Engineering Conference (APSEC)*, 2011, pp.266-273.

[19]. SAE AS-2C Architecture Description language Subcommittee, Embedded Computing Systems Committee, Aerospace Avionics Systems Division, “Architecture analysis and design language v2,” SAE Aerospace – an SAE International Groupe, AS5506A, 2009.

[20]. J. Delange, J. Hugues, L. Pautet et B. Zalila, “Code generation strategies from AADL architectural descriptions targeting the high integrity domain,” dans *4th European Congress ERTS*, 2008.

[21]. J. Chilenski, “Aerospace vehicle systems institute systems and software integration verification overview,” dans *AADL Safety and Security Modeling Meeting*, 2007.

[22]. T. Erkkinen et B. Potter, “Model-based design for DO-178B with qualified tools,” dans *AIAA Modeling and Simulation Technologies Conference*, 2009.

[23]. POK, “A partitioned POK operating system”, [En ligne], disponible: http://pok.tuxfamily.org/legal (Consulté: 20 Juin 2014).

[24]. B. Fons-Albert, H. Usach-Molina, J. Vila-Carbo et A. Crespo-Lorente, “Development of integrated modular avionics applications based on Simulink and XTRATUM,” dans *DASIA 2013 Data Systems in Aerospace*, 2013, pp.47-52.

[25]. R. Alena, J. Ossenfort, K. Laws et A. Goforth, Figueroa F. “Communications for integrated modular avionics,” dans *2007 IEEE Aerospace Conference*, 2006, pp.1-18.

[26]. Y. D. Li, W. Q. Cui, D. L. Li et R. Zhang, “Rearch Based on OSI model,” dans *2011 IEEE 3rd International Conference on Communication Software and Networks*, 2011, pp.554-557.

[27]. G. Raghav, S. Gopalswamy, J. Delange et J. Hugues, “Generation of AADL architecture consistent work products: Simulink behavioral models, and distributed embedded software using OCARINA”, dans *14th Conférence international on RELIABLE SOFTWARE TECHNOLOGIES*, ADA-Europe, 2009.

[28]. J. Hugues, T. Vergnaud et Z. Bechir, “OCARINA, A compiler for the AADL,” *École nationale supérieure des télécommunications*, Manuel usager, 2012.

[29]. J. Savard, L. Bao, G. Bois, J.F. Boland, “Model-Based design flow driven by integrated modular avionic simulations”, dans *SAE 2013 AeroTech Congress & Exhibition*, Montréal, 2013.

[30]. MathWorks, “Implement first-order representation of turbofan engine with controller,” [En ligne], disponible: http://www.mathworks.com/help/aeroblks/turbofanenginesystem.html (Consulté: 9 Juillet 2014).

[31]. MathWorks, “Compute Mach number using velocity and speed of sound,” [En ligne], disponible: http://www.mathworks.com/help/aeroblks/machnumber.html (Consulté: 9 Juillet 2014).

[32]. MathWorks, “Implement 1976 COESA lower atmosphere,” [En ligne], disponible: http://www.mathworks.com/help/aeroblks/coesaatmospheremodel.html (Consulté: 9 Juillet 2014).

[33]. MathWorks, “Compute dynamic pressure using velocity and air density,” [En ligne], disponible: http://www.mathworks.com/help/aeroblks/dynamicpressure.html (Consulté: 9 Juillet 2014).

[34]. MathWorks, “Calculate Earth’s magnetic field at specific location and time using World Magnetic Model,” [En ligne], disponible: http://www.mathworks.com/help/aeroblks/worldmagneticmodel2010.html (Consulté: 9 Juillet 2014).

[35]. MathWorks, “Implement 1984 World Geodetic System representation of Earth’s gravity,” [En ligne], disponible: http://www.mathworks.com/help/aeroblks/wgs84gravitymodel.html (Consulté: 9 Juillet 2014).

[36]. SYSGO Embedded Innovations, “PikeOS Hypervisor Eclipse based CODEO,” [En ligne], disponible: http://www.sysgo.com/en/products/pikeos-rtos-and-virtualization-concept/eclipse-based-codeo/ (Consulté: 27 Juillet 2014).

[37]. SAE, “Annex document F ARINC653”, *Society of Automotive Engineers*, AS5506/2, Annexe document, pp.42-60

[38]. B. Rajkumarsingh et S. Goolaup, “Modeling an Earthing System using Labview”, dans *Global Engineering Education Conference (EDUCON)* , IEEE, 2013, pp763-768.

[39]. J. Delange et L. Lec, “POK, an ARINC 653- compliant operating system released under the BSD license,” dans *13th Real-time Linux Workshop, Faculté de genie électrique de l’Université Technique Tchèque de Prague*, 2011. [En ligne], disponible : https://lwn.net/images/conf/rtlws-2011/proc/Delange_POK.pdf (Consulté: 4 Novembre 2014).

[40]. B. Zalila, L. Pautet, G. Lasnier et J. Hugues, “OCARINA- An environment for AADL models analysis and automatic code generation for high integrity applications,” dans *Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies*, France, 2009, pp.237-250

[41]. QEMU – Open Source Processor Emulator, “About QEMU,” [En ligne], disponible: http://wiki.qemu.org/Main_Page(Consulté: 7 Décembre 2014).

[42]. J. Rufino, J. Craveiro, T. Schoofs, C.Tatibana, et J. Windsor, “AIR Technology: a step towords ARINC653 in space,” dans *Proceedings of the DASIA 2009*: *Data Systems In Aerospace Conference*, Turquie, 2009.

[43]. G. Stringhamm, *Hardware/firmware interface design: best practices for improving embedded systems development*, Burlington, MA: Newnes, 2010.

[44]. GMV Innovating Solutions, "Configuima: System Description," [En ligne], disponible: http://www.gmv.com/en/aeronautics/products/configuima/ (Consulté: 19 Décembre 2014).

ANNEXE 1 – Le Code source en C généré par OCARINA

Fonction main.c généré vers POK

-- exemple: arinc653-queueing

```
#include "activity.h"
#include <arinc653/types.h>
#include "deployment.h"
#include <arinc653/queueing.h>
#include <arinc653/process.h>
#include <core/partition.h>
#include <arinc653/error.h>
#include <arinc653/partition.h>
/*****************************************************/
/*  This file was automatically generated by Ocarina */
/*  Do NOT hand-modify this file, as your            */
/*  changes will be lost when you re-run Ocarina     */
/*****************************************************/
PROCESS_ID_TYPE arinc_threads[POK_CONFIG_NB_THREADS];
QUEUING_PORT_ID_TYPE pr1_pdataout_id;
/***********/
/* main   */
/**********/

int main ()
{

  PROCESS_ATTRIBUTE_TYPE tattr;
  RETURN_CODE_TYPE ret;
  CREATE_QUEUING_PORT ("pr1_pdataout", sizeof (int), 2, SOURCE, FIFO, &(pr1_pdataout_id), &(ret));
  /*  This queueing port was mapped from an out event data port contained in */
  /*  the process. Itis used for inter-partition communication.*/
  tattr.ENTRY_POINT = thr_job;
  tattr.DEADLINE = 1000;
  tattr.PERIOD = 1000;
  tattr.STACK_SIZE = 40;
  tattr.TIME_CAPACITY = 1;
  CREATE_PROCESS (&(tattr), &(arinc_threads[1]), &(ret));
  /*  This thread was mapped from a thread component containedin this */
  /*  process. The function it executes is also generatedin the file */
  /*  activity.c.*/
  CREATE_ERROR_HANDLER (pok_error_handler_worker, 8192, &(ret));
  /*  One thread inside the partition can raise faults.We start the error */
  /*  handle to treat these potentialfaults.*/
  SET_PARTITION_MODE (NORMAL, &(ret));
  /*  Now, we created all resources of the process. Consequently,this thread */
  /*  will not be used any more and it will be keptin a dormant state. By */
  /*  doing that, we also allow one morethread in this partition*/
  return (0);
}
```

Fonction main.c généré vers SIMA

-- exemple: arinc653-queueing

```
#include "activity.h"
#include <a653.h>
#include "deploymentnew.h"
#include <kernel/deployment.h>
#include "pok_wrapper.h"
#include <stdio.h>
/*****************************************************/
/*  This file was automatically generated by Ocarina */
/*  Do NOT hand-modify this file, as your            */
/*  changes will be lost when you re-run Ocarina     */
/*****************************************************/
PROCESS_ID_TYPE arinc_threads[POK_CONFIG_NB_THREADS];
QUEUING_PORT_ID_TYPE pr1_pdataout_id;
/******************/
/* entry_point  */
/******************/

int entry_point ()
{

  PROCESS_ATTRIBUTE_TYPE tattr;
  RETURN_CODE_TYPE ret;
  CREATE_QUEUING_PORT ("pr1_pdataout", sizeof (int), 2, SOURCE, FIFO, &(pr1_pdataout_id), &(ret));
  ASSERT_RET(ret);
  /*  This queueing port was mapped from an out event data port contained in */
  /*  the process. Itis used for inter-partition communication.*/

  tattr.ENTRY_POINT = thr_job;
  tattr.BASE_PRIORITY = 10;
  tattr.DEADLINE = SOFT;
  tattr.PERIOD = 1000000000;
  tattr.STACK_SIZE = 40;
  tattr.TIME_CAPACITY = 900000000;
  CREATE_PROCESS (&(tattr), &(arinc_threads[1]), &(ret));
  START (arinc_threads[1], &(ret));
  ASSERT_RET(ret);
  /*  This thread was mapped from a thread component containedin this */
  /*  process. The function it executes is also generatedin the file */
  /*  activity.c.*/
  CREATE_ERROR_HANDLER (pok_error_handler_worker, 8192, &(ret));
  ASSERT_RET(ret);
  SET_PARTITION_MODE (NORMAL, &(ret));
  /*  Now, we created all resources of the process. Consequently,this thread */
  /*  will not be used any more and it will be keptin a dormant state. By */
  /*  doing that, we also allow one morethread in this partition*/
  return (0);
}
```

ANNEXE 2 – **Les 2 Fichiers XML générés par OCARINA**

Fichier a653.xml généré vers SIMA

-- exemple : arinc653-queueing

```
<ARINC_653_Module xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation=".Clearwater Version 1.1 .xsd"
ModuleName="Example ARINC 653 XML Instance for Projet AVIO509 by BAO Lin" ModuleVersion="15-May-2012">
 <System_HM_Table>
   <System_State_Entry SystemState="1" Description="mos is starting">
     <Error_ID_Level ErrorIdentifier="2" Description="Module Config" ErrorLevel="MODULE"/>
     <Error_ID_Level ErrorIdentifier="4" Description="Module Initialisation" ErrorLevel="MODULE"/>
     <Error_ID_Level ErrorIdentifier="5" Description="Segmentation Fault" ErrorLevel="MODULE"/>
     <Error_ID_Level ErrorIdentifier="7" Description="Illegal Instruction" ErrorLevel="MODULE"/>
     <Error_ID_Level ErrorIdentifier="8" Description="Numeric Error" ErrorLevel="MODULE"/>
     <Error_ID_Level ErrorIdentifier="9" Description="Stack Overflow" ErrorLevel="MODULE"/>
     <Error_ID_Level ErrorIdentifier="11" Description="Bad Opcode" ErrorLevel="MODULE"/>
     <Error_ID_Level ErrorIdentifier="12" Description="Power Failure" ErrorLevel="MODULE"/>
   </System_State_Entry>
   <System_State_Entry SystemState="2" Description="mos is executing">
     <Error_ID_Level ErrorIdentifier="5" Description="Segmentation Violation" ErrorLevel="MODULE"/>
     <Error_ID_Level ErrorIdentifier="6" Description="Timing Error" ErrorLevel="MODULE"/>
     <Error_ID_Level ErrorIdentifier="7" Description="Illegal Instruction" ErrorLevel="MODULE"/>
     <Error_ID_Level ErrorIdentifier="8" Description="Numeric Error" ErrorLevel="MODULE"/>
     <Error_ID_Level ErrorIdentifier="9" Description="Stack Overflow" ErrorLevel="MODULE"/>
     <Error_ID_Level ErrorIdentifier="11" Description="Bad Opcode" ErrorLevel="MODULE"/>
     <Error_ID_Level ErrorIdentifier="12" Description="Power Failure" ErrorLevel="MODULE"/>
   </System_State_Entry>
   <System_State_Entry SystemState="4" Description="pos system code is executing">
     <Error_ID_Level ErrorIdentifier="3" Description="Partition Config" ErrorLevel="PARTITION"/>
     <Error_ID_Level ErrorIdentifier="5" Description="Segmentation Fault" ErrorLevel="PARTITION"/>
     <Error_ID_Level ErrorIdentifier="6" Description="Timing Error" ErrorLevel="PARTITION"/>
     <Error_ID_Level ErrorIdentifier="7" Description="Illegal Instruction" ErrorLevel="PARTITION"/>
     <Error_ID_Level ErrorIdentifier="8" Description="Numeric Error" ErrorLevel="PARTITION"/>
     <Error_ID_Level ErrorIdentifier="9" Description="Stack Overflow" ErrorLevel="PARTITION"/>
     <Error_ID_Level ErrorIdentifier="11" Description="Bad Opcode" ErrorLevel="PARTITION"/>
     <Error_ID_Level ErrorIdentifier="12" Description="Power Failure" ErrorLevel="MODULE"/>
   </System_State_Entry>
   <System_State_Entry SystemState="5" Description="application code is executing">
     <Error_ID_Level ErrorIdentifier="5" Description="Segmentation Fault" ErrorLevel="PARTITION" ErrorCode="MEMORY_VIOLATION"/>
     <Error_ID_Level ErrorIdentifier="6" Description="Timing Error" ErrorLevel="PARTITION" ErrorCode="DEADLINE_MISSED"/>
     <Error_ID_Level ErrorIdentifier="7" Description="Illegal Instruction" ErrorLevel="PROCESS" ErrorCode="ILLEGAL_REQUEST"/>
     <Error_ID_Level ErrorIdentifier="8" Description="Numeric Error" ErrorLevel="PROCESS" ErrorCode="NUMERIC_ERROR"/>
     <Error_ID_Level ErrorIdentifier="10" Description="application error" ErrorLevel="PROCESS" ErrorCode="APPLICATION_ERROR"/>
     <Error_ID_Level ErrorIdentifier="9" Description="Stack Overflow" ErrorLevel="PROCESS" ErrorCode="NUMERIC_ERROR"/>
     <Error_ID_Level ErrorIdentifier="11" Description="Bad Opcode" ErrorLevel="PARTITION"/>
     <Error_ID_Level ErrorIdentifier="12" Description="Power Failure" ErrorLevel="MODULE"/>
   </System_State_Entry>
   <System_State_Entry SystemState="6" Description="RAISE APPLICATION ERROR in EH">
     <Error_ID_Level ErrorIdentifier="10" Description="application error" ErrorLevel="PARTITION"/>
   </System_State_Entry>
   <System_State_Entry SystemState="9" Description="mos HM is executing">
     <Error_ID_Level ErrorIdentifier="5" Description="Segmentation Fault" ErrorLevel="MODULE"/>
     <Error_ID_Level ErrorIdentifier="6" Description="Timing Error" ErrorLevel="MODULE"/>
     <Error_ID_Level ErrorIdentifier="7" Description="Illegal Instruction" ErrorLevel="MODULE"/>
     <Error_ID_Level ErrorIdentifier="8" Description="Numeric Error" ErrorLevel="MODULE"/>
     <Error_ID_Level ErrorIdentifier="9" Description="Stack Overflow" ErrorLevel="MODULE"/>
```

```
    <Error_ID_Level ErrorIdentifier="11" Description="Bad Opcode" ErrorLevel="MODULE"/>
    <Error_ID_Level ErrorIdentifier="12" Description="Power Failure" ErrorLevel="MODULE"/>
  </System_State_Entry>
</System_HM_Table>
<Module_HM_Table ModuleCallback="Module_HM_callback">
  <System_State_Entry SystemState="1" Description="mos is starting">
    <Error_ID_Action ErrorIdentifier="2" Action="SHUTDOWN"/>
    <Error_ID_Action ErrorIdentifier="4" Action="SHUTDOWN"/>
    <Error_ID_Action ErrorIdentifier="5" Action="SHUTDOWN"/>
    <Error_ID_Action ErrorIdentifier="7" Action="SHUTDOWN"/>
    <Error_ID_Action ErrorIdentifier="8" Action="SHUTDOWN"/>
    <Error_ID_Action ErrorIdentifier="9" Action="SHUTDOWN"/>
    <Error_ID_Action ErrorIdentifier="11" Action="SHUTDOWN"/>
    <Error_ID_Action ErrorIdentifier="12" Action="SHUTDOWN"/>
  </System_State_Entry>
  <System_State_Entry SystemState="2" Description="mos is execution">
    <Error_ID_Action ErrorIdentifier="5" Action="SHUTDOWN"/>
    <Error_ID_Action ErrorIdentifier="6" Action="SHUTDOWN"/>
    <Error_ID_Action ErrorIdentifier="7" Action="SHUTDOWN"/>
    <Error_ID_Action ErrorIdentifier="8" Action="SHUTDOWN"/>
    <Error_ID_Action ErrorIdentifier="9" Action="SHUTDOWN"/>
    <Error_ID_Action ErrorIdentifier="11" Action="RESET"/>
    <Error_ID_Action ErrorIdentifier="12" Action="RESET"/>
  </System_State_Entry>
  <System_State_Entry SystemState="4" Description="mos is execution">
    <Error_ID_Action ErrorIdentifier="12" Action="RESET"/>
  </System_State_Entry>
  <System_State_Entry SystemState="5" Description="application code is running">
    <Error_ID_Action ErrorIdentifier="12" Action="RESET"/>
  </System_State_Entry>
  <System_State_Entry SystemState="9" Description="mos HM is execution">
    <Error_ID_Action ErrorIdentifier="5" Action="SHUTDOWN"/>
    <Error_ID_Action ErrorIdentifier="6" Action="SHUTDOWN"/>
    <Error_ID_Action ErrorIdentifier="7" Action="SHUTDOWN"/>
    <Error_ID_Action ErrorIdentifier="8" Action="SHUTDOWN"/>
    <Error_ID_Action ErrorIdentifier="9" Action="SHUTDOWN"/>
    <Error_ID_Action ErrorIdentifier="11" Action="RESET"/>
    <Error_ID_Action ErrorIdentifier="12" Action="RESET"/>
  </System_State_Entry>
</Module_HM_Table>
<Partition PartitionName="pr2" PartitionIdentifier="1" EntryPoint="Initial" Criticality="LEVEL_A" SystemPartition="true">
  <Queuing_Port Direction="DESTINATION" Name="pdatain" MaxNbMessages="32" MaxMessageSize="1024"/>
</Partition>
<Partition PartitionName="pr1" PartitionIdentifier="2" EntryPoint="Initial" Criticality="LEVEL_A" SystemPartition="true">
  <Queuing_Port Direction="SOURCE" Name="pdataout" MaxNbMessages="32" MaxMessageSize="1024"/>
</Partition>
<Partition_Memory PartitionName="pr2" PartitionIdentifier="1">
  <Memory_Requirements SizeBytes="15000" Type="DATA" Access="READ-WRITE"/>
  <Memory_Requirements SizeBytes="10000" Type="CODE" Access="READ-WRITE"/>
  <Memory_Requirements SizeBytes="80000" Type="DATA" Access="WRITE"/>
</Partition_Memory>
<Partition_Memory PartitionName="pr1" PartitionIdentifier="2">
  <Memory_Requirements SizeBytes="80000" Type="CODE" Access="READ"/>
</Partition_Memory>
<Module_Schedule MajorFrameSeconds=" 2.0000">
  <Partition_Schedule PartitionName="pr2" PartitionIdentifier="1" PeriodSeconds=" 1.0000" PeriodDurationSeconds=" 1.0000">
    <Window_Schedule WindowStartSeconds=" 0.0000" WindowIdentifier="0" WindowDurationSeconds=" 1.0000" PartitionPeriodStart="true"/>
  </Partition_Schedule>
  <Partition_Schedule PartitionName="pr1" PartitionIdentifier="2" PeriodSeconds=" 1.0000" PeriodDurationSeconds=" 1.0000">
    <Window_Schedule WindowStartSeconds=" 1.0000" WindowIdentifier="1" WindowDurationSeconds=" 1.0000" PartitionPeriodStart="true"/>
  </Partition_Schedule>
</Module_Schedule>
```

```
</Module_Schedule>
<Connection_Table>
  <Channel ChannelIdentifier="1" ChannelName="1">
    <Source>
      <Standard_Partition PortName="pdataout" PartitionName="pr1" PartitionIdentifier="2"/>
    </Source>
    <Destination>
      <Standard_Partition PortName="pdatain" PartitionName="pr2" PartitionIdentifier="1"/>
    </Destination>
  </Channel>
</Connection_Table>
<Partition_HM_Table PartitionIdentifier="1" PartitionName="pr2" PartitionCallback="Partition_HM_callback">
  <System_State_Entry SystemState="4" Description="pos system code is executing">
    <Error_ID_Action ErrorIdentifier="3" Description="Partition Config" Action="IDLE"/>
    <Error_ID_Action ErrorIdentifier="5" Description="Segmentation Fault" Action="COLD_START"/>
    <Error_ID_Action ErrorIdentifier="6" Description="Timing Error" Action="IGNORE"/>
    <Error_ID_Action ErrorIdentifier="7" Description="Illegal Instruction" Action="WARM_START"/>
    <Error_ID_Action ErrorIdentifier="8" Description="Numeric Error" Action="WARM_START"/>
    <Error_ID_Action ErrorIdentifier="9" Description="Stack Overflow" Action="WARM_START"/>
    <Error_ID_Action ErrorIdentifier="11" Description="Bad Opcode" Action="WARM_START"/>
  </System_State_Entry>
  <System_State_Entry SystemState="5" Description="application code is executing">
    <Error_ID_Action ErrorIdentifier="5" Description="Segmentation Fault" Action="IDLE"/>
    <Error_ID_Action ErrorIdentifier="6" Description="Timing Error" Action="WARM_START"/>
    <Error_ID_Action ErrorIdentifier="11" Description="Bad Opcode" Action="WARM_START"/>
  </System_State_Entry>
  <System_State_Entry SystemState="6" Description="RAISE_APPLICATION_ERROR in error handler">
    <Error_ID_Action ErrorIdentifier="10" Description="Application Error" Action="WARM_START"/>
  </System_State_Entry>
</Partition_HM_Table>
<Partition_HM_Table PartitionIdentifier="2" PartitionName="pr1" PartitionCallback="Partition_HM_callback">
  <System_State_Entry SystemState="4" Description="pos system code is executing">
    <Error_ID_Action ErrorIdentifier="3" Description="Partition Config" Action="IDLE"/>
    <Error_ID_Action ErrorIdentifier="5" Description="Segmentation Fault" Action="COLD_START"/>
    <Error_ID_Action ErrorIdentifier="6" Description="Timing Error" Action="IGNORE"/>
    <Error_ID_Action ErrorIdentifier="7" Description="Illegal Instruction" Action="WARM_START"/>
    <Error_ID_Action ErrorIdentifier="8" Description="Numeric Error" Action="WARM_START"/>
    <Error_ID_Action ErrorIdentifier="9" Description="Stack Overflow" Action="WARM_START"/>
    <Error_ID_Action ErrorIdentifier="11" Description="Bad Opcode" Action="WARM_START"/>
  </System_State_Entry>
  <System_State_Entry SystemState="5" Description="application code is executing">
    <Error_ID_Action ErrorIdentifier="5" Description="Segmentation Fault" Action="IDLE"/>
    <Error_ID_Action ErrorIdentifier="6" Description="Timing Error" Action="WARM_START"/>
    <Error_ID_Action ErrorIdentifier="11" Description="Bad Opcode" Action="WARM_START"/>
  </System_State_Entry>
  <System_State_Entry SystemState="6" Description="RAISE_APPLICATION_ERROR in error handler">
    <Error_ID_Action ErrorIdentifier="10" Description="Application Error" Action="WARM_START"/>
  </System_State_Entry>
</Partition_HM_Table>
</ARINC_653_Module>
```

Fichier sima.xml généré vers SIMA

-- exemple : arinc653-queueing

```
<mos system="Test" a653_config="a653.xml" Startup="/bin/sh ./samples/sima-generated/mos/mos.sh" partitions="2" granularity="100000">
 <Output type="fifo" path="./samples/sima-generated/mos/fifo_mos"/>
 <Partition PartitionIdentifier="1" PartitionName="pr2" SharedMemory="50010" Startup="/bin/sh ./samples/sima-generated/partitions/pr2/pr2.sh" visible="true">
   <Output type="fifo" path="./samples/sima-generated/partitions/pr2/fifo_pr2"/>
   <Transport Start="all" End="all" StartDurationSeconds="0.01" EndDurationSeconds="0.01"/>
   <Destination_Port Name="pdatain" Type="UDP" IP="127.0.0.1" Port="12351"/>
 </Partition>
 <Partition PartitionIdentifier="2" PartitionName="pr1" SharedMemory="60010" Startup="/bin/sh ./samples/sima-generated/partitions/pr1/pr1.sh" visible="true">
   <Output type="fifo" path="./samples/sima-generated/partitions/pr1/fifo_pr1"/>
   <Transport Start="all" End="all" StartDurationSeconds="0.01" EndDurationSeconds="0.01"/>
 </Partition>
</mos>
```

ANNEXE 3 – **Le fichier pok_wrapper.c**

Fichier pok_wrapper.h

```
#ifndef __POK_WRAPPER_H_
#define __POK_WRAPPER_H_

#include <stdio.h>
#include <a653.h>


//typedef RETURN_CODE_TYPE pok_ret_t;
typedef OPERATING_MODE_TYPE pok_partition_mode_t;


//#define but pok_ret_t
#define ASSERT_RET(ret) if (ret != NO_ERROR) { printf ("ASSERTION FAILED,ret=%d, file=%s, line=%d\n", ret, __FILE__, __LINE__);}

#define POK_PARTITION_MODE_INIT_COLD    COLD_START
#define POK_PARTITION_MODE_INIT_WARM    WARM_START
#define POK_PARTITION_MODE_NORMAL       NORMAL
#define POK_PARTITION_MODE_IDLE         IDLE
#define POK_PARTITION_MODE_RESTART      WARM_START
#define POK_PARTITION_MODE_STOPPED      IDLE


/* RESTART A PROCESS */
pok_ret_t pok_thread_restart (PROCESS_ID_TYPE id);

pok_ret_t pok_partition_set_mode (const pok_partition_mode_t mode);

/* Function that does nothing aside from quitting the error handler. No logging. */
#define pok_error_ignore(error_id, thread_id)   return (void*)(NULL)

#endif
```

Fichier pok_wrapper.c

```
#include "pok_wrapper.h"

/* RESTART A PROCESS */
pok_ret_t pok_thread_restart (PROCESS_ID_TYPE id){
        RETURN_CODE_TYPE ret;
        STOP(id, &ret);
        /* printf("STOPPED process\n"); */
        START(id, &ret);
        /* printf("STARTED process\n"); */
}

pok_ret_t pok_partition_set_mode (const pok_partition_mode_t mode){
        pok_ret_t ret;
        SET_PARTITION_MODE(mode, &ret);
        return ret;
}
```

ANNEXE 4 – **Port de communication TCP/IP**

Fichier wrapper.h

```
#ifndef __Client__
#define __Client__

#include"capteurs.h"

//#include <WinSock.h>  // older version, limited version

typedef double real_T;

// External inputs
typedef struct {
  real_T altitude;
  real_T speed[3];
  real_T latitude;
  real_T longtitude;
  real_T throttle;
} ExtU_capteurs_T;

// External outputs
typedef struct {
  real_T T;
  real_T a;
  real_T Ps;
  real_T Intensity;
  real_T Pd;
  real_T Mach;
  real_T Thrust;
  real_T Fuel;
  real_T Mf[3];
  real_T HI;
  real_T TI;
  real_T G[3];
} ExtY_capteurs_T;


#define PORT 9000
#define LISTENQ 20

#define SOCKET_STREAM 1
#define AF_INET 2
#define IPPROTO_TCP 6
#define SCK_VERSION1            0x0101
#define SCK_VERSION2            0x0202

typedef struct _DataPackage{
       ExtU_capteurs_T DataInSensor;  // Generated code Input type, defined in capteurs.h
       ExtY_capteurs_T DataOutSensor; // Generated code Output type, defined in capteurs.h
       double distance[2]; // 0: Total Distance 1: Distance Run
       unsigned long int ID; // package ID
}DataPackage;
```

```
	bool ConnectToHost(char IPaddr[20]);

	void SendData(char* DatatoSend, unsigned int size);

	void CloseSendConnection();

	bool ListenClient(void);

	void receData(DataPackage &recepackage);

#endif
```

Fichier wrapper.c

```
#include <stdio.h>
#include <WinSock2.h>
#include "wrapper.h"


#pragma comment(lib, "ws2_32.lib")


int listenfd, connectfd, sockfd;
struct sockaddr_in servaddr, cliaddr;
HWND hwnd;

SOCKET SocketArray[WSA_MAXIMUM_WAIT_EVENTS]; // socket array
WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS]; // event array
WSANETWORKEVENTS NewworkEvents;  // network events
DWORD EventTotal = 0, Index = 0;
char buffer[sizeof(DataPackage)];
WSAEVENT NewEvent;

/*****************************************************************************
 Send data package part
*****************************************************************************/

bool ConnectToHost(char IPaddr[20]){
	struct sockaddr_in servaddr;
	int ret;

	WSADATA wsadata;
	ret = WSAStartup(0x0202, &wsadata);

	if(ret)
		return false;

	if(wsadata.wVersion != 0x0202){
		WSACleanup();
		return false;
	}

	memset(&servaddr,'0', sizeof(servaddr));
```

```
	//bzero(&servaddr, sizeof(servaddr));
	servaddr.sin_family = AF_INET;
	servaddr.sin_port = htons(PORT);
	servaddr.sin_addr.s_addr = inet_addr(IPaddr);

	if((sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == INVALID_SOCKET){ //
create a socket, AF_INET is the defaut familiy address
		fprintf(stderr, "ERROR1: Create Client Socket Error!");
		return false;
	}

	if(connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == SOCKET_ERROR)
{
		fprintf(stderr, "ERROR2: Client Connect Error!");
		return false;
	}else
		return true;

}

void SendData(char* DatatoSend, unsigned int size){ // send data function
	send(sockfd, DatatoSend, size, 0);
}

void CloseSendConnection(){
	if(sockfd)
		closesocket(sockfd);
	WSACleanup(); // used to clear up the winsock
}


/*********************************************************************************
 Receive data package part
*********************************************************************************/

bool ListenClient(void){
	WSADATA wsadata;

	int error = WSAStartup(0x0202, &wsadata);
	if(error) {
		printf("Winsock couldn't be started!\n");
		return false;
	}

	if(wsadata.wVersion != 0x0202){
		WSACleanup();
		return false;
	}

	memset(&servaddr, 0, sizeof(servaddr));
	servaddr.sin_family = AF_INET;
	servaddr.sin_port = htons(PORT);
	servaddr.sin_addr.s_addr = INADDR_ANY;

	if((listenfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == INVALID_SOCKET){
	// create socket
```

```
                fprintf(stderr, "Create Server Socket Error!");
                return false;
        }

        if(bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) == SOCKET_ERROR){
                fprintf(stderr, "Server Bind Error!"); // if we try to bind the same socket
for more than once
                return false;
        }

        NewEvent = WSACreateEvent();
        error = WSAEventSelect(listenfd, NewEvent, FD_ACCEPT|FD_CLOSE);
        if(error == 0){
                printf("Association of event objet and network events is successful!\n");
        }else{
                printf("Events association error: %d\n", error);
                return false;
        }

        if(listen(listenfd, LISTENQ) == SOCKET_ERROR){
                fprintf(stderr, "Server Listen Error!");
                return false;
        }else{
                printf("Listening on socket ...\n");
        }

        SocketArray[EventTotal] = listenfd;
        EventArray[EventTotal] = NewEvent;
        EventTotal ++;
}

// receive data package
void receData(DataPackage &recepackage){
                //printf("Inside boucle!\n");
               Index = WSAWaitForMultipleEvents(EventTotal, EventArray, false, WSA_INFINITE,
false);
                WSAEnumNetworkEvents(SocketArray[Index - WSA_WAIT_EVENT_0], EventArray[Index
- WSA_WAIT_EVENT_0], &NewworkEvents);

                if(NewworkEvents.lNetworkEvents & FD_ACCEPT){// FD_ACCEPT
                        printf("Message1: Connection request was made!\n");
                        if(NewworkEvents.iErrorCode[FD_ACCEPT_BIT] != 0){
                                printf("FD_Accept error!\n");
                                exit(1);
                        }

                        int cliaddrlen;
                        cliaddrlen = sizeof(cliaddr);
                        connectfd = accept(SocketArray[Index - WSA_WAIT_EVENT_0], (struct
sockaddr*)&cliaddr, &cliaddrlen);

                        if(EventTotal > 64){
                                printf("Too many events!\n");
                                closesocket(connectfd);
                                exit(0);
                        }
```

```
                NewEvent = WSACreateEvent();
                WSAEventSelect(connectfd, NewEvent, FD_READ|FD_WRITE|FD_CLOSE);
                SocketArray[EventTotal] = connectfd;
                EventArray[EventTotal] = NewEvent;
                EventTotal ++;
                printf("Socket %d connected!\n", connectfd);
        }// End FD_accept

        if(NewworkEvents.lNetworkEvents & FD_READ){//FD_READ
                //printf("Message2: New message to read!\n");

                if(NewworkEvents.iErrorCode[FD_READ_BIT] != 0){
                        printf("FD_Read error!\n");
                        exit(2);
                }

                memset(buffer, 0, sizeof(buffer));
                memset(&recepackage, 0, sizeof(DataPackage));
               recv(SocketArray[Index - WSA_WAIT_EVENT_0], buffer, sizeof(buffer),0);
                memcpy(&recepackage, buffer, sizeof(DataPackage));

              //printf("ID: %d\t Data:%f\n",recepackage.ID, recepackage.DataSensor);


        }// End FD_Read

        if(NewworkEvents.lNetworkEvents & FD_WRITE){//FD_Write
                printf("Message3: Ready to write!\n");

                if(NewworkEvents.iErrorCode[FD_READ_BIT] != 0){
                        printf("FD_write error!\n");
                        exit(3);
                }

        }//End FD_write

        if(NewworkEvents.lNetworkEvents & FD_CLOSE){//FD_close
                printf("Message4: Close the sockets!\n");

                if(NewworkEvents.iErrorCode[FD_CLOSE_BIT] != 0){
                        printf("FD_close error!\n");
                        exit(4);
                }

                closesocket(SocketArray[Index - WSA_WAIT_EVENT_0 ]);

                WSACloseEvent(EventArray[Index - WSA_WAIT_EVENT_0 ]);
                for(int i=Index - WSA_WAIT_EVENT_0; i <EventTotal; i++ ){
                        EventArray[i] = EventArray[i+1];
                        SocketArray[i] = SocketArray[i+1];
                }
                EventTotal --;
        }//End FD_close

}
```

ANNEXE 5 – **L'exemple du modèle AADL textuel**

```
package sima_app
public
with Data_Model;
with ARINC653;

-- 1. Declaration of packages
data integer
properties
  Data_Model::Data_Representation => integer;
end integer;

data Basetype
properties
  Data_Model::Data_Representation => character;
  Data_Model::Number_Representation => Unsigned;
  Source_Data_Size => 1 Bytes;
end Basetype;

data FuelPacket
properties
  Data_Model::Data_Representation => Array;
  Data_Model::Base_Type => (classifier(sima_app::Basetype));
  Data_Model::Dimension => (24);
end FuelPacket;

data EnviPacket
properties
  Data_Model::Data_Representation => Array;
  Data_Model::Base_Type => (classifier(sima_app::Basetype));
  Data_Model::Dimension => (104);
end EnviPacket;

data FlightPacket
properties
  Data_Model::Data_Representation => Array;
  Data_Model::Base_Type => (classifier(sima_app::Basetype));
  Data_Model::Dimension => (72);
end FlightPacket;

data serveurPacket
properties
  Data_Model::Data_Representation => Array;
  Data_Model::Base_Type => (classifier(sima_app::Basetype));
  Data_Model::Dimension => (208);
end serveurPacket;

-- 2.Declaration and implementation of the vitual processors (partitions)
virtual bus secure_bus
properties
   Required_Connection_Quality_Of_Service => (SecureDelivery);
end secure_bus;

virtual bus medium_bus
```

```
properties
   Required_Connection_Quality_Of_Service => (SecureDelivery);
end medium_bus;

virtual bus common_bus
properties
   Required_Connection_Quality_Of_Service => (SecureDelivery);
end common_bus;

virtual processor partition
properties
   ARINC653::HM_Errors => (Partition_Init);
   ARINC653::HM_Actions => (Partition_Restart);
end partition;

virtual processor implementation partition.serveur
properties
   Provided_Virtual_Bus_Class =>
               (classifier (sima_app::secure_bus));
end partition.serveur;

virtual processor implementation partition.fuel
properties
   Provided_Virtual_Bus_Class =>
               (classifier (sima_app::medium_bus));
end partition.fuel;

virtual processor implementation partition.environment
properties
   Provided_Virtual_Bus_Class =>
               (classifier (sima_app::medium_bus));
end partition.environment;

virtual processor implementation partition.flight
properties
   Provided_Virtual_Bus_Class =>
               (classifier (sima_app::secure_bus));
end partition.flight;

virtual processor implementation partition.view
properties
   Provided_Virtual_Bus_Class =>
               (classifier (sima_app::common_bus));
end partition.view;

-- 3. Implementation of processor
processor ppc
end ppc;

processor implementation ppc.impl
subcomponents
   partition_serveur : virtual processor partition.serveur;
   partition_fuel : virtual processor partition.fuel;
   partition_environment : virtual processor partition.environment;
   partition_flight : virtual processor partition.flight;
```

```
   partition_view : virtual processor partition.view;
properties
   ARINC653::Module_Major_Frame => 1000 ms;
   ARINC653::Partition_Slots => (200 ms, 200 ms, 200 ms, 200 ms, 200 ms);
   ARINC653::Slots_Allocation => ( reference (partition_serveur),reference
(partition_fuel), reference (partition_environment),
reference(partition_flight), reference(partition_view));
end ppc.impl;

-- 4.Declare and implement the processes, the corresponding threads and
subprograms.

--*****************************subprogram*****************************
subprogram sub_serveur
features
   serveur_output : out parameter serveurPacket;
properties
   source_text => ("../../../app_serveur.o");
   source_language => C;
   source_name => "app_serveur";
end sub_serveur;

subprogram sub_fuel
features
   fuel_input : in parameter serveurPacket;
   fuel_output : out parameter FuelPacket;
properties
   source_text => ("../../../app_fuel.o");
   source_language => C;
   source_name => "app_fuel";
end sub_fuel;

subprogram sub_environment
features
   envi_input : in parameter serveurPacket;
   envi_output : out parameter EnviPacket;
properties
   source_text => ("../../../app_environment.o");
   source_language => C;
   source_name => "app_environment";
end sub_environment;

subprogram sub_flight
features
   flight_input : in parameter serveurPacket;
   flight_output : out parameter FlightPacket;
properties
   source_text => ("../../../app_flight.o");
   source_language => C;
   source_name => "app_flight";
end sub_flight;

subprogram sub_view
features
   fuelview_input : in parameter FuelPacket;
```

```
   enviview_input : in parameter EnviPacket;
   flightview_input : in parameter FlightPacket;
properties
   source_name => "app_view";
   source_language => C;
   source_text => ("../../../app_view.o");
end sub_view;

--*********************************Threads*****************************
thread thread_serveur
features
   tdataout_serveur : out event data port serveurPacket{ARINC653::Timeout =>
10 ms;};
properties
   Dispatch_Protocol => Periodic;
   Period => 1000 Ms;
   Deadline => 990 Ms;
   Compute_Execution_Time => 0 ms .. 1 ms;
   Recover_Execution_Time => 10 ms .. 20 ms;
   Source_Data_Size => 100 bytes;
   Source_Stack_Size => 100 bytes;
   Source_Code_Size => 100 bytes;
   ARINC653::HM_Errors => (Stack_Overflow);
   ARINC653::HM_Actions => (Ignore);
end thread_serveur;

thread implementation thread_serveur.impl
calls
   call1 : { serveurpspg : subprogram sub_serveur;};
connections
   parameter serveurpspg.serveur_output -> tdataout_serveur;
end thread_serveur.impl;

thread thread_fuel
features
   tdatain_fuel : in event data port serveurPacket{ARINC653::Timeout => 10
ms;};
   tdataout_fuel : out event data port FuelPacket {ARINC653::Timeout => 10
ms;};
properties
   Dispatch_Protocol => Periodic;
   Compute_Execution_Time => 0 ms .. 1 ms;
   Period => 1000 Ms;
   Deadline => 990 Ms;
   ARINC653::HM_Errors => (Numeric_Error);
   ARINC653::HM_Actions => (Partition_Restart);
   Source_Data_Size => 100 bytes;
   Source_Stack_Size => 100 bytes;
   Source_Code_Size => 100 bytes;
end thread_fuel;

thread implementation thread_fuel.impl
calls
   call1 : { fuelpspg : subprogram sub_fuel;};
connections
```

```
   parameter tdatain_fuel -> fuelpspg.fuel_input;
   parameter fuelpspg.fuel_output -> tdataout_fuel;
end thread_fuel.impl;

thread thread_environment
features
   tdatain_envi : in event data port serveurPacket{ARINC653::Timeout => 10
ms;};
   tdataout_envi : out event data port EnviPacket{ARINC653::Timeout => 10 ms;};
properties
   Dispatch_Protocol => Periodic;
   Period => 1000 Ms;
   Deadline => 990 Ms;
   Source_Data_Size => 100 bytes;
   Source_Stack_Size => 100 bytes;
   Source_Code_Size => 100 bytes;
   Compute_Execution_Time => 0 ms .. 1 ms;
   ARINC653::HM_Errors => (Numeric_Error);
   ARINC653::HM_Actions => (Partition_Restart);
end thread_environment;

thread implementation thread_environment.impl
calls
   call1 : { envipspg : subprogram sub_environment;};
connections
   parameter tdatain_envi -> envipspg.envi_input;
   parameter envipspg.envi_output -> tdataout_envi;
end thread_environment.impl;

thread thread_flight
features
   tdatain_flight : in event data port serveurPacket{ARINC653::Timeout => 10
ms;};
   tdataout_flight : out event data port FlightPacket{ARINC653::Timeout => 10
ms;};
properties
   Dispatch_Protocol => Periodic;
   Period => 1000 Ms;
   Deadline => 990 Ms;
   Compute_Execution_Time => 0 ms .. 1 ms;
   Recover_Execution_Time => 10 ms .. 20 ms;
   Source_Data_Size => 100 bytes;
   Source_Stack_Size => 100 bytes;
   Source_Code_Size => 100 bytes;
   ARINC653::HM_Errors => (Stack_Overflow);
   ARINC653::HM_Actions => (Ignore);
end thread_flight;

thread implementation thread_flight.impl
calls
   call1 : { flightpspg : subprogram sub_flight;};
connections
   parameter tdatain_flight -> flightpspg.flight_input;
   parameter flightpspg.flight_output -> tdataout_flight;
end thread_flight.impl;
```

```
thread thread_view
features
   tdatain_fuelview : in event data port FuelPacket {ARINC653::Timeout => 10
ms;};
   tdatain_enviview : in event data port EnviPacket {ARINC653::Timeout => 10
ms;};
   tdatain_flightview : in event data port FlightPacket {ARINC653::Timeout =>
10 ms;};
properties
   Dispatch_Protocol => Periodic;
   Recover_Execution_Time => 10 ms .. 20 ms;
   Compute_Execution_Time => 0 ms .. 1 ms;
   Period => 1000 Ms;
   Deadline => 990 Ms;
   Source_Data_Size => 100 bytes;
   Source_Stack_Size => 100 bytes;
   Source_Code_Size => 100 bytes;
   ARINC653::HM_Errors => (Stack_Overflow);
   ARINC653::HM_Actions => (Ignore);
end thread_view;

thread implementation thread_view.impl
calls
   call1 : { viewpspg : subprogram sub_view;};
connections
   parameter tdatain_fuelview -> viewpspg.fuelview_input;
   parameter tdatain_enviview -> viewpspg.enviview_input;
   parameter tdatain_flightview -> viewpspg.flightview_input;
end thread_view.impl;

--************************processes*************************
process process_serveur
features
   pdataout_serveur : out event data port
serveurPacket{Allowed_Connection_Binding_Class => (classifier
(sima_app::medium_bus)); Queue_Size => 2;};
properties
   Source_Code_Size => 10 KByte;
   Source_Data_Size => 15 KByte;
end process_serveur;

process implementation process_serveur.impl
subcomponents
   thread_serveur : thread thread_serveur.impl;
connections
   port thread_serveur.tdataout_serveur -> pdataout_serveur;
end process_serveur.impl;

process process_fuel
features
   pdatain_fuel : in event data port serveurPacket
{Allowed_Connection_Binding_Class => (classifier (sima_app::medium_bus));
Queue_Size => 2;};
```

```
   pdataout_fuel : out event data port FuelPacket
{Allowed_Connection_Binding_Class => (classifier (sima_app::medium_bus));
Queue_Size => 2;};
properties
   Source_Code_Size => 10 KByte;
   Source_Data_Size => 15 KByte;
end process_fuel;

process implementation process_fuel.impl
subcomponents
   thread_fuel : thread thread_fuel.impl;
connections
   port pdatain_fuel -> thread_fuel.tdatain_fuel;
   port thread_fuel.tdataout_fuel -> pdataout_fuel;
end process_fuel.impl;

process process_environment
features
   pdatain_envi : in event data port serveurPacket
{Allowed_Connection_Binding_Class => (classifier (sima_app::medium_bus));
Queue_Size => 2;};
   pdataout_envi : out event data port EnviPacket
{Allowed_Connection_Binding_Class => (classifier (sima_app::medium_bus));
Queue_Size => 2;};
properties
   Source_Code_Size => 10 KByte;
   Source_Data_Size => 15 KByte;
end process_environment;

process implementation process_environment.impl
subcomponents
   thread_environment : thread thread_environment.impl;
connections
   port pdatain_envi -> thread_environment.tdatain_envi;
   port thread_environment.tdataout_envi -> pdataout_envi;
end process_environment.impl;

process process_flight
features
   pdatain_flight : in event data port serveurPacket
{Allowed_Connection_Binding_Class => (classifier (sima_app::medium_bus));
Queue_Size => 2;};
   pdataout_flight : out event data port FlightPacket
{Allowed_Connection_Binding_Class => (classifier (sima_app::medium_bus));
Queue_Size => 2;};
properties
   Source_Code_Size => 10 KByte;
   Source_Data_Size => 15 KByte;
end process_flight;

process implementation process_flight.impl
subcomponents
   thread_flight : thread thread_flight.impl;
connections
   port pdatain_flight -> thread_flight.tdatain_flight;
```

```
   port thread_flight.tdataout_flight -> pdataout_flight;
end process_flight.impl;

process process_view
features
   pdatain_fuelview : in event data port FuelPacket
{Allowed_Connection_Binding_Class => (classifier (sima_app::medium_bus));
Queue_Size => 2;};
   pdatain_enviview : in event data port EnviPacket
{Allowed_Connection_Binding_Class => (classifier (sima_app::medium_bus));
Queue_Size => 2;};
   pdatain_flightview : in event data port FlightPacket
{Allowed_Connection_Binding_Class => (classifier (sima_app::medium_bus));
Queue_Size => 2;};
properties
   Source_Code_Size => 10 KByte;
   Source_Data_Size => 15 KByte;
end process_view;

process implementation process_view.impl
subcomponents
   thread_view : thread thread_view.impl;
connections
   port pdatain_fuelview -> thread_view.tdatain_fuelview;
   port pdatain_enviview -> thread_view.tdatain_enviview;
   port pdatain_flightview -> thread_view.tdatain_flightview;
end process_view.impl;

-- 5.Declare and implement the partition memory
memory partitionmemory
properties
   Word_Count => 128000;
   ARINC653::Memory_Kind => memory_code;
   ARINC653::Access_Type => read;
end partitionmemory;

memory mainmemory
end mainmemory;

memory implementation mainmemory.impl
subcomponents
   mem_serveur: memory partitionmemory
               {ARINC653::Memory_Kind => memory_data;
               ARINC653::Access_Type => write;};
   mem_fuel: memory partitionmemory
               {ARINC653::Memory_Kind => memory_data;
               ARINC653::Access_Type => write;};
   mem_environment: memory partitionmemory
               {ARINC653::Memory_Kind => memory_data;
               ARINC653::Access_Type => write;};
   mem_flight: memory partitionmemory
               {ARINC653::Memory_Kind => memory_data;
               ARINC653::Access_Type => write;};
   mem_view: memory partitionmemory
               {ARINC653::Memory_Kind => memory_code;
```

```
                ARINC653::Access_Type => read;};
end mainmemory.impl;

-- 6.Implement the system
system node
end node;

system implementation node.impl
subcomponents
   avionic : processor ppc.impl;
   serveur : process process_serveur.impl;
   fuel : process process_fuel.impl;
   envi : process process_environment.impl;
   flight : process process_flight.impl;
   view : process process_view.impl;
   mem : memory mainmemory.impl;

connections
   port serveur.pdataout_serveur ->
fuel.pdatain_fuel{Allowed_Connection_Binding_Class => (classifier
(sima_app::secure_bus));};
   port serveur.pdataout_serveur ->
envi.pdatain_envi{Allowed_Connection_Binding_Class => (classifier
(sima_app::secure_bus));};
   port serveur.pdataout_serveur ->
flight.pdatain_flight{Allowed_Connection_Binding_Class => (classifier
(sima_app::secure_bus));};
   port fuel.pdataout_fuel->
view.pdatain_fuelview{Allowed_Connection_Binding_Class => (classifier
(sima_app::secure_bus));};
   port envi.pdataout_envi->
view.pdatain_enviview{Allowed_Connection_Binding_Class => (classifier
(sima_app::secure_bus));};
   port flight.pdataout_flight->
view.pdatain_flightview{Allowed_Connection_Binding_Class => (classifier
(sima_app::secure_bus));};
properties
   actual_processor_binding =>
      (reference (avionic.partition_serveur)) applies to serveur;
   actual_processor_binding =>
      (reference (avionic.partition_fuel)) applies to fuel;
   actual_processor_binding =>
      (reference (avionic.partition_environment)) applies to envi;
   actual_processor_binding =>
      (reference (avionic.partition_flight)) applies to flight;
   actual_processor_binding =>
      (reference (avionic.partition_view)) applies to view;

   actual_memory_binding =>
      (reference (mem.mem_serveur)) applies to serveur;
   actual_memory_binding =>
      (reference (mem.mem_fuel)) applies to fuel;
   actual_memory_binding =>
      (reference (mem.mem_environment)) applies to envi;
   actual_memory_binding =>
```

```
      (reference (mem.mem_flight)) applies to flight;
   actual_memory_binding =>
      (reference (mem.mem_view)) applies to view;

end node.impl;

end sima_app;
```